INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48108
JAIN, Nirmal, 1945-

USE OF PROGRAM STRUCTURE INFORMATION IN VIRTUAL MEMORY MANAGEMENT.

University of Hawaii, Ph.D., 1975
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48108
USE OF PROGRAM STRUCTURE INFORMATION IN
VIRTUAL MEMORY MANAGEMENT

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN ELECTRICAL ENGINEERING

MAY 1975

By

Nirmal Jain

Dissertation Committee:
Art Y. Lew, Chairman
W. Wesley Peterson
Franklin F. Kuo
W. Wayne Lichtenberger
Rahul Chattergy
ABSTRACT

In this dissertation, we first obtain an a priori model of program behavior based upon localities. We define a locality as a set of pages which are used together, and we obtain a priori estimates of these by using information about program connectivity. Then using this structural information, we obtain feasible algorithms for the pagination, replacement, and memory allocation problems in virtual memory computer systems.

For the pagination problem, a primary shortcoming in most earlier solutions is the assumption of single page localities implicitly made in adopting the criterion of minimizing the number of interpage references. We instead adopt a space-time cost criterion which includes the cost of references made outside the resident set (the current locality), as well as the cost of main memory occupied at any time. We amend another major shortcoming by treating data as a class separate from the instructions. Then for the minimization of space-time cost, and for segregation of data and instructions, the pagination procedure we recommend is to paginate the localities onto a minimum number of pages, separately for instructions and data.

With regard to demand-paging replacement, we first develop "optimal" algorithms, optimal in the sense of minimizing the expected number of page faults, under Markovian and locality assumptions on program execution. LPR(N), based upon knowledge of transition probabilities among pages, replaces a page with the least probability of reference along all paths of length < N from the currently referenced page. INR, based upon structural rather than probabilistic information, replaces a page which
has the maximum interval (path length) of next reference from the currently referenced page. Finally, we show that exemptive algorithms can be derived from the INR algorithm, and such algorithms, in general, provide better performance than their underlying historical replacement algorithms. Exemptive algorithms divide the resident set into an exempt set, e.g. the set of pages which have a path of length < N from the currently referenced page, and a nonexempt set. The page selected to be replaced is the lowest ranking (using reference history, i.e. the historical information) page in the latter set.

Memory allocation is divided into three sub-problems: partitioning, locality size estimation, and locality membership determination. In partitioning, i.e. division of main memory resources among programs, we show for non-identical programs that an "optimal" partition must at least equal the locality size estimate and that "excess" memory must be distributed in proportion to variances in locality size. For locality size estimation, i.e. estimation of size of main memory required for efficient execution of a program, we employ Bayesian decision techniques to combine structural and historical information. We establish the validity of using Bayesian estimates of locality size through simulation. Finally, the problem of memory membership, i.e. determination of what pages to keep as members of the locality, is considered for a pre-loading environment. (For demand paging it entails using an appropriate replacement algorithm.) The a priori information on localities is used to load an entire locality as a batch at the time it is entered. This saves a large number of page faults which would occur if no a priori information were used. We have shown that implementation of such a scheme is feasible with respect to the BCC 500 computer system.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ABSTRACT</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 GENERAL</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Introduction to Virtual Memory Systems</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Optimization Problems in Virtual Memory Systems</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3 Program Graph--Definitions</td>
<td>8</td>
</tr>
<tr>
<td>1.1.4 Program Behavior and Locality Model</td>
<td>12</td>
</tr>
<tr>
<td>1.2 REORDERING AND PAGINATION</td>
<td>16</td>
</tr>
<tr>
<td>1.2.1 Earlier Work</td>
<td>16</td>
</tr>
<tr>
<td>1.2.2 Need for Specialized Treatment of Data</td>
<td>19</td>
</tr>
<tr>
<td>1.2.3 Importance of Reordering</td>
<td>20</td>
</tr>
<tr>
<td>1.2.4 Present Work</td>
<td>22</td>
</tr>
<tr>
<td>1.3 REPLACEMENT PROBLEM</td>
<td>24</td>
</tr>
<tr>
<td>1.3.1 Historical Replacement Algorithms</td>
<td>25</td>
</tr>
<tr>
<td>1.3.2 Replacement Algorithms Using Probabilistic Information</td>
<td>26</td>
</tr>
<tr>
<td>1.3.3 Present Work</td>
<td>27</td>
</tr>
<tr>
<td>1.4 MEMORY ALLOCATION PROBLEM</td>
<td>30</td>
</tr>
<tr>
<td>1.4.1 Earlier Work</td>
<td>30</td>
</tr>
<tr>
<td>1.4.2 Present Work--Optimal Partitioning, Estimation of Locality Size, Determination of Membership</td>
<td>32</td>
</tr>
<tr>
<td>1.4.3 Implementation on BCC 500</td>
<td>35</td>
</tr>
<tr>
<td>CHAPTER 2. REORDERING AND PAGINATION</td>
<td>36</td>
</tr>
<tr>
<td>2.1 SEGREGATION OF DATA AND INSTRUCTION</td>
<td>36</td>
</tr>
<tr>
<td>2.2 REORDERING AND PAGINATION OF INSTRUCTION NODES</td>
<td>39</td>
</tr>
<tr>
<td>2.2.1 Ordering</td>
<td>39</td>
</tr>
<tr>
<td>2.2.2 Pagination</td>
<td>45</td>
</tr>
<tr>
<td>2.3 REORDERING AND PAGINATION OF DATA NODES</td>
<td>54</td>
</tr>
<tr>
<td>2.3.1 Page Fault Analysis for Data Nodes</td>
<td>57</td>
</tr>
<tr>
<td>2.3.2 Reordering Procedures</td>
<td>62</td>
</tr>
<tr>
<td>2.3.3 Pagination Procedure</td>
<td>74</td>
</tr>
<tr>
<td>2.4 COLLECTION OF A PRIORI INFORMATION</td>
<td>80</td>
</tr>
<tr>
<td>2.4.1 Determination of Localities</td>
<td>80</td>
</tr>
<tr>
<td>2.4.2 Mean and Variance in Locality Size</td>
<td>85</td>
</tr>
<tr>
<td>2.4.3 Locality Membership</td>
<td>87</td>
</tr>
<tr>
<td>2.4.4 Discussion</td>
<td>88</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1 Page Fault Comparison for Instruction Pagination (with respect to Kernighan's)</td>
<td>51</td>
</tr>
<tr>
<td>2.2.2 Page Fault Comparison for Instruction Pagination (with respect to Baer and Caughey's)</td>
<td>51</td>
</tr>
<tr>
<td>2.3.1 Space-time Costs for Example 1</td>
<td>67</td>
</tr>
<tr>
<td>2.3.2 Space-time Costs for Example 2</td>
<td>67</td>
</tr>
<tr>
<td>2.3.3 Page Fault Comparison for Data Pagination</td>
<td>78</td>
</tr>
<tr>
<td>2.4.1 Estimated Execution Time Table for SCRs in Figure 1.1.1</td>
<td>89</td>
</tr>
<tr>
<td>3.3.1 Page Fault Comparison of Replacement Algorithms</td>
<td>121</td>
</tr>
<tr>
<td>4.4.1 Simulation Results for the Pre-loading Algorithms</td>
<td>173</td>
</tr>
<tr>
<td>4.4.2 Loading Costs and Drum Service Rates for the Pre-loading Algorithms</td>
<td>173</td>
</tr>
</tbody>
</table>
## LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 A FORTRAN Program &amp; the Program Graph</td>
<td>10</td>
</tr>
<tr>
<td>1.1.2 The Connectivity Matrix</td>
<td>11</td>
</tr>
<tr>
<td>1.1.3 Locality Model</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 An Example for Segregated and Non-segregated Pagination</td>
<td>37</td>
</tr>
<tr>
<td>2.1.2 Another Example for Segregated and Non-segregated Pagination</td>
<td>37</td>
</tr>
<tr>
<td>2.2.1 A Program Example</td>
<td>42</td>
</tr>
<tr>
<td>2.2.2 Kernighan's Example and its Connectivity Matrix</td>
<td>42</td>
</tr>
<tr>
<td>2.2.3 Reduced Graph for Baer and Caughey's Example and its Connectivity Matrix</td>
<td>42</td>
</tr>
<tr>
<td>2.2.4 Program Node and Page Graphs for Kernighan's Example</td>
<td>46</td>
</tr>
<tr>
<td>2.2.5 Program Page Graphs for Baer and Caughey's Example</td>
<td>46</td>
</tr>
<tr>
<td>2.2.6 Program Node Trace</td>
<td>51</td>
</tr>
<tr>
<td>2.2.7 Program Page Traces</td>
<td>51</td>
</tr>
<tr>
<td>2.2.8 A Program Node Graph</td>
<td>53</td>
</tr>
<tr>
<td>2.2.9 Corresponding Page Graphs</td>
<td>53</td>
</tr>
<tr>
<td>2.2.10 Page Fault vs. Resident Set Size Graph</td>
<td>53</td>
</tr>
<tr>
<td>2.3.1 Inter-Locality Reference Frequency Matrices</td>
<td>61</td>
</tr>
<tr>
<td>2.3.2 Node Trace - Including Data References</td>
<td>77</td>
</tr>
<tr>
<td>2.3.3 Page Traces</td>
<td>77</td>
</tr>
<tr>
<td>2.3.4 Page Fault vs. Resident Set Size Graph</td>
<td>78</td>
</tr>
<tr>
<td>3.2.1 An Illustrative Example</td>
<td>106</td>
</tr>
<tr>
<td>3.3.1 Reduced Substate Transition Diagram</td>
<td>117</td>
</tr>
<tr>
<td>4.2.1 Timing Diagram for Simulation</td>
<td>149</td>
</tr>
<tr>
<td>4.2.2 Timing Diagram for Simulation of Dynamic Algorithm</td>
<td>149</td>
</tr>
<tr>
<td>4.2.3 Timing Diagram for Simulation of Bayesian Algorithm</td>
<td>149</td>
</tr>
<tr>
<td>4.2.4 Memory Exceedence vs. Memory Size Graphs</td>
<td>153</td>
</tr>
<tr>
<td>4.2.5 Memory Exceedence vs. Memory Size Graphs</td>
<td>154</td>
</tr>
<tr>
<td>4.2.6 Memory Exceedence vs. Memory Size Graphs</td>
<td>155</td>
</tr>
<tr>
<td>4.2.7 Memory Exceedence vs. Memory Size Graphs</td>
<td>156</td>
</tr>
<tr>
<td>4.2.8 Page Fault vs. Memory Size Graphs</td>
<td>153</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Page Fault vs. Memory Size Graphs</td>
</tr>
<tr>
<td>4.2.10</td>
<td>Page Fault vs. Memory Size Graphs</td>
</tr>
<tr>
<td>4.2.11</td>
<td>Page Fault vs. Memory Size Graphs</td>
</tr>
<tr>
<td>4.2.12</td>
<td>Memory Exceedence vs. Memory Size Graphs for Varying Locality Duration</td>
</tr>
<tr>
<td>4.2.13</td>
<td>Page Fault vs. Memory Size Graphs for Varying Locality Duration</td>
</tr>
<tr>
<td>D-1</td>
<td>Address Translation via PM</td>
</tr>
<tr>
<td>D-2</td>
<td>Address Translation via PMT and CHT</td>
</tr>
<tr>
<td>D-3</td>
<td>A PMT and a MAP Entry</td>
</tr>
<tr>
<td>D-4</td>
<td>An APT Entry</td>
</tr>
<tr>
<td>D-5</td>
<td>CB Contents</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 GENERAL

1.1.1 Introduction to Virtual Memory Systems

The past few years have seen a significant increase in interest of the computing community in virtual memory computer systems. A virtual memory system may be defined generally as a computer system in which there is a distinction between the logical address generated by a program and the physical address for some storage device from which the information is actually obtained. A general definition may thus also include conventional file systems where data sets are referenced by name rather than directly in terms of device and position information. The common form of virtual memory systems, however, is one in which memory addresses are translated (relocated) dynamically by the system. A simple system of this type is a computer where a single relocation constant is used to obtain the effective address. A more general form permits the virtual storage to be split into blocks (generally known as pages), each with its own dynamically changable relocation constant. It is this latter form of virtual memory system that concerns us in the present work.

The virtual memory system assumed here is a computer system with two levels of memory hierarchy. The two levels are fast main memory (monolithic or core) which is immediately addressable by the processor(s), and the slower secondary (auxiliary) memory which is not
directly addressable. The auxiliary memory assumed here is a drum memory, as opposed to disk memories or bulk memories (ECS or LCS) of very large capacity. The drum is assumed to be of moderately high rotational speed with a high transfer rate, in keeping with present day systems. Two-level memory systems have been designed to cut down on the cost of main memory, which continues to be one of the most expensive parts of modern day computer systems, while at the same time providing memory access times close to those of fast memory. This is made possible by keeping only the active parts of programs in the main memory and keeping the rest on the slower memory device. Such an organization offers a number of advantages.

From a programmer's point of view, the major advantages are:

- large amounts of virtual memory are provided at low cost and high speed.
- the programmer does not have to worry about memory management since these tasks are delegated to the operating system.
- better development of application programs is provided by allowing unrestricted use of address space.

Since only the active portions (subsets that are actually in use) of the program occupy main storage at any given time, it is possible to give the programmer much more logical address space than would otherwise be possible. Thus he can avoid having to work with overlay structures that are often necessary in conventional systems. The overlaying of information still takes place in the virtual memory systems, but it is handled automatically by the system and is trans-
parent to the programmer. For a programmer who is developing applications programs that must run in a broad range of system configurations, with virtual memory systems, only a single version of a program needs to be developed that will run in any amount of main storage.

Virtual memory systems also offer significant system advantages:

- the utilization efficiency of main memory is high.
- potential for multiprogramming is increased.

In conventional systems the main memory may be underutilized because of fragmentation associated with the allocation of large continuous blocks. In virtual memory systems, the blocks into which the storage is divided can be allocated discontiguously. The resulting reduction in fragmentation, combined with the fact that only active portions of virtual address space are maintained in the main memory at any given time, can substantially increase main memory utilization and hence the degree of multiprogramming.

There are two common approaches to virtual storage organization:

- paging--virtual memory is divided into fixed length blocks and the main memory is allocated similarly.
- segmentation--virtual memory is divided into variable length segments that may or may not be subdivided into pages for allocation to physical storage.

Paging is generally logically transparent to the programmer, and may be considered solely a storage management mechanism. Segmentation may or may not be visible to the programmer. For details of these systems see Denning [1]. We are concerned in the present work with paged virtual memory systems only. In such systems the page serves as a unit of
information storage as well as of transfer between main and auxiliary storage. A page of information may reside in any of the available blocks in the main memory. The actual address (physical) is obtained through the address translation hardware.

The virtual memory systems considered here may operate either in a batch multiprogramming environment or in a time-sharing environment. In a batch multiprogramming environment, a program once loaded in the main memory remains there until execution is completed. A number of such programs may coexist in main memory sharing the resources of the system such as to improve utilization efficiency of all system components. Time-sharing systems on the other hand have a fast turnover of programs in the main memory, due to the fast response requirement for such programs. Once a program is loaded in the main memory, it may execute for only a fixed time (quantum) before being removed (swapped out) from the main memory.

In this work, frequent reference is made to one virtual memory computer system, the BCC 500, whenever illustrative examples of important concepts appear warranted. Descriptions of the BCC 500 may be found in [2,3,60] and a summary is given in Appendix D.

1.1.2 Optimization Problems in Virtual Memory Systems

The basic limitation of availability of main memory resources creates a number of problems for virtual memory computer systems. One problem is that of memory allocation: in particular, the determination of the amount of main memory ("resident set size") to be allotted to various users, and the determination of the number of users ("degree of
multiprogramming") which will be allocated resources simultaneously. With respect to paging, the equivalent problem is that of deciding how many pages of main memory should be allocated to a program.

A second problem is to determine which pages should occupy the main memory space allotted to a program, i.e. to determine resident sets as a function of time. This can be divided into two subproblems, namely,

- the fetch problem.
- the replacement problem.

The fetch problem is that of deciding what pages to load and when. For demand paging a page is loaded when it is referenced but not in the main memory. For pre-loading the problem is to anticipate what pages may be needed during the following interval of time and to load these pages. The replacement problem is one of deciding what page to replace when another page not in the main memory is desired.

A third problem arises because the virtual memory as well as the physical storage is divided into equal sized blocks, i.e. pages. Since the program (and data) is not partitioned in a similar manner, the problem is one of deciding how to distribute the program code (and data) among the pages of virtual storage. This is generally known as the pagination problem.

One of the main directions of research relating to virtual memory systems is that of obtaining optimal solutions to these problems. A fundamental strategy in all paging algorithms is to minimize page traffic during execution periods of a program, or more specifically to minimize the number of page faults. Other optimality criteria (e.g.
throughput, processor utilization, response time) are often favored (especially in scheduling literature), but page-faulting is the most commonly adopted criterion in paging literature [27–29] (in part for tractability reasons). Each page fault causes the program to wait for the referenced page to be loaded, and during this time the main memory occupied by this program is tied up unproductively degrading main memory utilization. For single processor systems each page fault further deteriorates system efficiency because the processor must thereupon do some systems work (e.g. to initiate load and unload operations, choose a proper page to replace, etc.). Higher page fault rates also introduce more page traffic between memories (causing congestion) and lead to high CPU idleness (due to no eligible process being available to run). We thus adopt minimization of the number of page faults as a basic criterion for obtaining more efficient algorithms for solutions of the aforementioned problems. However, for some purposes, a modified criterion is used which represents the practical situation more realistically, e.g. for pagination algorithms we use a space-time cost (which includes page fault count) as the criteria for minimization.

A number of earlier solutions to paging problems make use of probabilistic information for obtaining optimal (or efficient) algorithms. We argue against the exclusive use of probabilistic information because:

1) The optimality of the algorithms depends upon a priori knowledge of reference probabilities, and their distributions. If these probabilities cannot be estimated with a certain degree of accuracy, there is no point in using
these algorithms. Since reference probabilities may be
time-varying and data dependent, they need to be re-
estimated for every run of the program and within each
run, which makes implementation of these algorithms
expensive. If ensemble or long term probabilities are
used instead, they may not represent the probabilities
for a particular execution appropriately. The cost of
runs to obtain these probabilities may itself be prohibi-
tive in some cases, while in others it may not be possible
to obtain this information at all.

ii) Furthermore, if a program is data dependent, as most
programs are, simple probabilistic models (e.g. station-
ary homogeneous Markov chains) may not be realistic.

Some authors use frequency of reference information obtained from
erlier executions (of programs) for designing efficient paging algo-
rithms. This may also be criticized on the basis of data dependence
of programs: for different data, acquired frequency values may be
sufficiently different so as to nullify any gains made through use of
such information. There is also some question as to the practicality
of making trial runs.

Instead of probabilistic information, we make use of the knowledge
of program structure in the design of our algorithms. The program
structure information may be obtained during compilation, and some
compilers (e.g. IBM's FORTRAN H [19] and PL/I Optimizing [20]) already
do so. Since the explicit structure of a program does not change during
the course of its execution, such information can be used with more
confidence than any probabilistic information. We also use the history of reference during current execution in conjunction with structural information in designing algorithms. The structural information is mainly used to provide a priori estimates about the program requirements and the historical information is used to update or aid these estimates.

1.1.3 Program Graph

The structural information used is the connectivity information for the program graph. The program graph represents the reference paths within a program. The graphical model used to represent programs is basically that of Lowe [4]. A program graph consists of the following:

- a directed sub-graph for representation of flow of control among the instruction nodes.
- an undirected sub-graph for representation of connectivity between instruction and data nodes.

The instruction nodes and data nodes are defined as follows:

**Instruction node:** an instruction node may consist of a number of instructions such that every instruction node is entered at the top and exited only at the bottom, i.e. the only branch instruction out of the node is the last instruction.

**Data node:** a data node consists of a matrix, data structure, or one or more simple variables such that each variable in the same data node is referenced by the same set of instruction nodes. (We place no restriction on the size of a data node.)

A program then consists of \( m \) instruction nodes and \( n \) data nodes.

We can now give a formal definition for a program graph. We restrict
ourselves to serially executable programs.

**Program graph:** a program graph is defined as a 6-tuple \((I,D,E_I,E_D,\alpha_I,\alpha_T)\) where,

- \(I\) is a set of instruction nodes; \(\{\alpha_1, \alpha_2, \ldots, \alpha_m\}\).
- \(D\) is a set of data nodes; \(\{\beta_1, \beta_2, \ldots, \beta_n\}\).
- \(E_I \subseteq I \times I\) is a set of directed arcs connecting instruction nodes.
- \(E_D\) is a set of undirected arcs connecting instruction nodes with data nodes.
- \(\alpha_I \in I\), the initial node of program.
- \(\alpha_T \in I\), the terminal node of program.

The **connectivity matrix** for a program graph is a matrix which represents a one-step connectivity among nodes of the graph. The connectivity matrix \(C\) may be partitioned into two matrices \(N\) and \(Q\) as follows:

\[
C = N \cdot Q
\]

where

\[
n_{ij} = \begin{cases} 
1, & \text{if instruction node } \alpha_j \text{ is an immediate successor of } \alpha_i. \\
0, & \text{otherwise.} 
\end{cases}
\]

\[
q_{ij} = \begin{cases} 
1, & \text{if data node } \beta_j \text{ is connected to instruction node } \alpha_i. \\
0, & \text{otherwise.} 
\end{cases}
\]

An example of a program graph and its connectivity matrix are given in Figures 1.1.1 and 1.1.2, which corresponds to the program
READ (5,5) K1, I1, I2, I3
IF (K1) 10, 20, 300
STOP
K1 = K1 + 1
I1 = I1 + 1
IF (I1) 25, 30, 30
I2 = I2 + 1
IF (I2) 30, 35, 35
I3 = I3 + 1
IF (I3) 35, 40, 40
IF ((I1+I2+I3).LT. 15) GO TO 23
DO 4000 I = 1, 50
M(I) = I*I1 + K1
M(I) = 2*M(I)
IF (M(I).LT. 25) GO TO 3995
CONTINUE
IF (K1-15) 23, 400, 400
DO 4000 I = 1, 50
CONTINUE
IF (K1-50) 300, 300, 10
DO 195 J = 1, 10
DO 190 I = K1, 50
N(I, J) = 2*I/15*J
CONTINUE
END
The Connectivity Matrix

**Figure 1.1.2**

A = \{1, 2, 3\}, B = \{1, 2, 4\}, D = \{1, 4, 6\} etc.

Locality Sequence: ABDCBE.....

**Figure 1.1.3**
example considered in Baer and Caughey [5]. (The '*' in the connectivity matrix corresponds to reference to a matrix element, and the dashed nodes and lines correspond to data nodes and their connections to instruction nodes.)

Finally, we give a definition of important sub-graphs of program graphs, namely strongly connected regions.

**Strongly Connected Region (SCR):** a subgraph of a digraph is a strongly connected region if there exists an oriented path between any two nodes of the subgraph. (SCRs can only exist in the instruction part of a program graph.)

A **maximal** SCR (MSCR) is an SCR not contained within another one. A **reduced** graph is one with an SCR replaced by a single node with connectivity relations with other nodes retained; the nodes of the SCR are then said to be "merged."

1.1.4 **Program Behavior and Locality Model**

For the design of optimal algorithms for virtual memory systems, one factor which is of prime importance is the estimation of program behavior. If program behavior is known accurately, paging schemes can be designed optimally. However, precise estimation of program behavior is generally not possible because of its dependence on data. Also, program behavior is among the least understood aspects of computer system design and analysis, and an area of a fair amount of speculation. And yet we need to model program behavior if we are to have a sound basis on which to predict a program's future needs, as required for algorithms we develop in the present work. We adopt the **locality model**
introduced by Denning [6].

The locality model assumes in essence that programs have the property that small subsets (the localities) of a program maintain control for long periods of time. Denning justifies this assumption by the observation that programmers frequently use sequential and looping structures and tend to concentrate on small parts of large problems for fairly long intervals of time. While not all programs may possess the locality property, the model has proven itself to be of value in past studies.

The locality model for program behavior has been formalized by Spirn, Denning, and Savage [7], and is based on the following:

• a program distributes its references non-uniformly over its pages, some pages being favored over others.
• the membership of the set of favored pages tends to change slowly in time.
• two reference string segments are highly correlated when the interval between them is small, and tend to become uncorrelated as the interval between them becomes large.

They contend that program execution may be represented as execution of localities, as shown in Figure 1.1.3. Associated with a program are a number of localities, denoted A, B, C, D, E in the figure, each consisting of a set of pages (e.g. locality A consists of pages 1, 2, 3). Each locality, by definition, keeps control for a period before control moves to another locality. During that period of time the paging activity will be minimal if all the information belonging to that locality is maintained in physical storage; in other words, resident sets should be
chosen to correspond to localities.

The problem remains, then, of estimating the size and membership of the "localities." Denning's working set algorithm [8] determines estimates using the historical information obtained during the execution of a program; working sets are essentially dynamic estimates of localities. We shall instead make locality estimates using a priori information on the strongly connected regions in the instruction graph and the data needed by them. Since the SCRs are the sub graphs in the instruction graph which may be expected to maintain control for long periods of time, it is logical to base locality estimates on the SCRs. However, each SCR will not be a locality because it may only consist of a few nodes and maintain control for a short time. We thus restrict our estimates of localities to be associated with those SCRs which satisfy the following conditions:

I. The size of a locality (instructions + data) is smaller than the maximum allocation size allowed for the program.

II. The expected execution time of the instructions in the locality is above a certain threshold value (determined by the waiting time to load pages, the quantum, etc.).

Each locality thus consists of two parts:

- instruction sub-locality.
- data sub-locality.

The instruction sub-localities correspond to SCRs which satisfy conditions I and II above, and the data sub-localities are the corresponding data referenced by these SCRs. In the graph of Figure 1.1.1, the sets of instruction nodes \{14,15,16,17\} and \{21,22,23\} represent
possible instruction sub-localities. The set of data nodes \{C50, C15, J, C2, K1, I\} is the data sub-locality associated with the latter instruction sub-locality.
1.2 REORDERING AND PAGINATION PROBLEM

The problem of distribution of code and data into pages may be subdivided into two subproblems:

- Reordering—restructuring of sections of program and data.
- Pagination—division of reordered code and data into pages.

We first review the work done already in this area and then go on to outline the approach we have taken.

1.2.1 Earlier Work

Much of the earlier work in the pagination problem, Kral [9], Lowe [10], and Kernighan [11], for example, is based on the assumption of a Markov model for the program with fixed transition probabilities (or reference frequencies) between nodes. A critical review of this work may be found in [61]. Briefly, they treat pagination as a covering problem. An "optimal" cover of a program graph is a collection B of disjoint sets, each element set of B being a set of nodes belonging to a page, such that references between sets of B are minimal. Kral shows that the problem may be reduced to a problem in pseudo-boolean programming and also to an integer linear programming problem. Solution to either problem is time consuming and hence impractical. Lowe solves the optimal problem using backtrack programming techniques, but even for this method the solution is unrealistic because the computational load grows greatly as the number of nodes increase. (The maximum number of operations required to obtain the optimal solution is proportional to m!, where m is the total number of nodes in the program
Kernighan imposes a linear ordering constraint on nodes, and the nodes on any page are constrained to be contiguous. The only degree of freedom, here, is in selecting the break points between pages. He uses dynamic programming to determine the break points, so as to minimize the number of inter-page transfers. Although the linearity constraint here leads to a comparatively efficient pagination algorithm, the constraint is unduly restrictive.

Yue and Wong [12] consider pagination assuming equal sized nodes and the knowledge of request probabilities for each node. Based on a supposition of statistically independent and identically distributed memory references, they develop a pagination scheme which optimizes the number of distinct pages referenced between references to the same page. Their scheme, however, fails to take into account the effect of serial dependence (cf. Section 5.4 [12]).

Ramamoorthy [13] was the leading advocate of the use of structural information for pagination. He suggests that the pagination problem may be solved by:

- Partitioning the program graph into maximal SCR}s and linear link sub graphs.
- For each maximal SCR of size greater than a page size, deleting (i.e. forcing a page boundary at) that branch which has the minimal frequency of execution.

Lowe [14], VerHoef [15], and Baer and Caughey [5] have proposed pagination algorithms based on same underlying principles. Lowe, and Baer and Caughey propose to preserve the minimal length cycles (loops) as much as possible. The idea is not to break any inner-most loops.
This approach is heuristically sound and we make use of this idea for pagination of instruction nodes of the program. VerHoef proposes to merge all nodes of loops smaller than a page size into single nodes before pagination. Lowe, and Baer and Caughey also employ this in their pagination algorithms.

The algorithms reviewed above have the following shortcomings:

• The pagination algorithms discussed above have been designed under the implicit assumption that only one page of the program can reside in the main memory at any time. In reality, a program almost always has more than one of its pages in the memory (the resident set) during execution. The interpage references for pages within the resident set should not be counted in the cost of pagination, as they result in no page traffic. If the resident sets can be determined a priori, the reordering and pagination algorithms should be directed towards reducing interpage references between pages in the resident set and the remaining pages.

• The assumption of linearity is particularly restrictive. Hatfield and Gerald [16] and Ferrari [17] have shown that the fact that program nodes can be reordered leads to large savings in the number of page faults. A pagination optimal under the assumption of a linearly ordered set of nodes will generally not be optimal otherwise.

• The replacement algorithm and pagination are closely related. Optimality of a pagination algorithm would depend
on the replacement algorithm and vice versa. For minimizing the page traffic during execution of the program, pagination and replacement algorithms should be designed in conjunction, a fact neglected entirely by earlier schemes.

• The assumption of probabilistic information has the deficiencies cited earlier (cf. Section 1.1.2).

1.2.2 A Need for Specialized Treatment of Data

A major shortcoming of the algorithms considered previously is their nontreatment of the data portions of programs. Data normally forms a large portion of the total space requirement of a program, and data activity is frequently quite high during execution of the program. A question that arises then is whether data nodes be treated as instruction nodes, or as a separate class of nodes. If the latter, then the problem is whether the basis for reordering and pagination of data nodes should be the same as that for instruction nodes.

In view of the fact that for most programs, data references are made more often than instruction references, we feel that data nodes should be treated as a special class of nodes. Another point which favors their special treatment is that data nodes may be connected to a number of instruction nodes which may be far apart, while instruction nodes can normally be separated into groups (i.e. SCRs) with minimal interaction. Also, all data nodes connected to an instruction node may be needed each time that instruction node is referenced, but this is not so for the instruction nodes connected to the same instruction node. Because of their essential difference, we develop pagination algorithms
separately for instruction nodes and data nodes. The special nature of data nodes is also depicted in the program graph, defined earlier. The connections between instruction nodes and data nodes are represented via special un-directed arcs.

Lowe [10] is the only one to our knowledge who explicitly provides a solution to the problem of data pagination separate from that of instruction pagination. He gives a search procedure to obtain the data pagination, based upon the knowledge of node transition probabilities. This is unsatisfactory for several reasons:

- the dependence on probability information.
- the search procedure is too expensive to implement for practical systems.
- the replacement environment is not considered.

1.2.3 Importance of Reordering

Another factor which has not been emphasized by most of the earlier schemes is the need for reordering of program nodes. The importance of reordering has been demonstrated most strikingly by Hatfield and Gerald [16]. They divide programs into "sectors" (similar to nodes) and devise ways of ordering sectors so as to minimize the interpage activity. The improvements obtained by them, measured in terms of total number of page faults, over unordered programs were of the order of 2 to 10. Some important considerations in their reordering procedure include:

- bringing parts of the program close together in space if they are required close together in time.
• filling each page with sectors which communicate more (i.e. have greater nearness requirements) with each other than with the rest of the program.
• minimizing links between any page out of physical memory and all pages in it.
• preventing frequently used code from extending over a page boundary.

Hatfield and Gerald base their reordering decisions on the values of the inter-sector reference frequency matrix obtained through a trial execution of the program. Ferrari [17] has also proposed a restructuring scheme. He makes use of a program node trace to find the "critical" page references (i.e. the references which cause a fault operating in a working set environment [8]), and paginates so as to minimize the number of such critical references. This is achieved by including the node(s) causing critical references into one of the pages of the working set (at the time of critical references). The main drawback of this method is that it reduces the page fault count at the cost of increased average working set size. A deficiency of both schemes (Hatfield and Gerald's, and Ferrari's) is the reliance of the restructuring algorithm on availability of a "typical" trial run of the program. Bovet and Estrin [18] also give a scheme for pagination (and memory allocation) based upon knowledge of frequency of references. They propose determination of these frequencies through extensive analysis of program graph using Markovian assumptions and assuming some a priori probabilities of transition among nodes. We criticized these assumptions earlier (Section 1.1.2).
1.2.4 Present Work

The implication of previous discussions is that better reordering and pagination algorithms need to be designed which are to some extent more realistic in their assumptions. To this end we propose that reordering and pagination be based upon structural information. We recommend the use of program connectivity information for this purpose. Such information is used to locate the sections of the program with the greatest nearness requirements, the loops and the SCRs in the instruction graph, for example. For reordering of data nodes, the connectivity between instruction and data nodes and the instruction SCRs are used to determine what data nodes should be put together.

We also deviate from the usual implicit assumption of single page resident set sizes by considering a set of nodes which are needed together and try to paginate such that they remain together on pages. This was first suggested by Lew [61]. The critical working set scheme of Ferrari [17] is the only such other attempt, to our knowledge. Our effort here is to optimize the use of main memory through minimizing the space allocated as well as by minimizing the time wasted due to page faults. We thus develop a joint space-time criteria which includes the cost of space wasted due to page faults as well as cost of main memory during execution. This can be obtained from the space-time cost, C, defined by Belady and Kuehner [21], during time interval (0,T), as

\[ C = \int_{0}^{T} s(\tau) d\tau, \]
where \( s(\tau) \) is the amount of main memory allotted to the program at real-time \( \tau \). In a multiprogramming system the interval \((0,T)\) would also include the periods of inactivity during which the program is blocked and waiting for demanded pages to be loaded in. Thus the cost may be written as,

\[
C = \int_0^{T_{exec}} s(t) \, dt + \sum_{i=1}^{x} s(t_i) \cdot t_w
\]  

(1.2.1)

where \( T_{exec} = \) time actually spent in execution,

\( t = \) virtual-time,

\( x = \) total number of page faults encountered,

\( s(t_i) = \) memory space occupied at \( i^{th} \) page fault,

\( t_w = \) average wait time to load a demanded page.

We observe that, in addition to minimizing the number of page faults \( x \), this cost criterion suggest that methods for reducing resident set sizes \( s \) should be sought. (The critical working set method of Ferrari only aims at minimizing a part of the second term in (1.2.1).) We discuss this further in Chapter 2.
1.3 REPLACEMENT PROBLEM

The replacement problem, in a demand paging context is that of deciding upon each page fault, which page residing in the main memory is to be removed. If an incorrect choice is made, and the page replaced is needed shortly, the program will page fault prematurely, thus increasing the number of page faults experienced by the system. The increase in the number of page faults leads to a deterioration in service of virtual memory systems, both for batch multiprogramming and time-sharing systems. For batch systems, during a page fault the memory is occupied unproductively by the program thus leading to lower main memory utilization. Besides it leads to an increased load on the processor and the secondary memory. For time-sharing systems, a high page fault rate leads to large number of swapping operations (if a program is unloaded on every page fault), or to a lower main memory utilization (if the program waits for demanded page to be loaded). Thus minimization of page faults and design of optimal replacement algorithms are very important.

Intuitively, a replacement algorithm should choose a page for replacement whose next reference is not expected for a long time. Belady [22] has formulated an optimal algorithm (proved optimal by Mattson et.al. [29]) in which the page to be referenced furthest away in the future is the one to be replaced. This algorithm is unrealizable for real systems because the optimal replacement decisions rely on knowledge of future page references. For realizable algorithms, only the present and the past history should be used in making a replacement decision. In the following, we consider some commonly
used replacement schemes based upon the history of program references, and later we discuss some "optimal" replacement algorithms based upon a priori information.

1.3.1 Historical Replacement Algorithms

The performance of a replacement algorithm depends upon its ability to predict future program behavior with accuracy. The various algorithms proposed base their predictions on different types of information available about the program. Most replacement algorithms, heretofore proposed and in common use, base their predictions upon the historical information, e.g.

- LRU—least recently used replacement algorithm.
- LFU—least frequently used replacement algorithm.
- FIFO—first in first out replacement algorithm.

The LRU replacement algorithm assumes that the page which was least recently used is also the one which is least likely to be referenced amongst pages in memory, and hence should be replaced. Similarly LFU bases its decision on the frequency of use of a page. FIFO replaces the page that entered main memory first, among the pages of the resident set. It has been shown, through various experiments and simulations, that none of these algorithms always produces the lowest number of page faults. This fact has led some authors [23] to adopt an adaptive strategy which switches between replacement algorithms depending upon which algorithm would have been optimal if it had been in use just previously.
Other variations of historical replacement algorithms have also been considered, the most prominent ones being the working set strategy [8] and the page fault frequency (PFF) [24] replacement algorithm. The working set strategy can be likened to an LRU scheme with variable size of resident set. The working sets are also determined by historical information, and are used in effect to predict future program behavior. The PFF replacement algorithm is similar to the working set algorithm except for parameter $T$, the window size, which is fixed for working set algorithm but is a variable for the PFF algorithm. Multics [25] implements a variation of working set strategy where working sets are estimated based on a partial page-fault trace. Tenex [26] implements a scheme similar to the PFF replacement scheme.

1.3.2 Replacement Algorithms Using Probabilistic Information

A second type of information commonly used for replacement algorithms is probabilistic information, usually in the form of a transition probability matrix for the program graph. The program page reference strings are assumed to exhibit statistical stationarity properties so that a stationary Markov model may be assumed for the reference string generation process, i.e. for the program. Under this and a few additional assumptions, a number of "optimal" and sub-optimal replacement algorithms have been proposed. One such algorithm is $A_o$ [27], optimal under the assumption that each page of the program has a stationary reference probability. It is easy to show that for realistic programs $A_o$ is not optimal. Ingargiola and Korsh [28] give a method to determine the optimal replacement algorithm.
for a program with Markovian assumptions. For this and the locality assumption, we derive an optimal replacement algorithm LPR, which replaces the page with the least probability of reference from the currently referenced page. The optimality in the above schemes is defined in terms of minimization of expected page-fault cost. Some sub-optimal replacement algorithms include [29]:

- LTP—which replaces the page with least transition probability from currently referenced page.
- LNR—which replaces a page with longest expected time of next reference.

Bovet and Estrin [18] give a replacement algorithm which uses a mixture of frequency and probability information to derive an "interaction" criterion, and the page with minimum interaction is replaced.

1.3.3 Present Work

Replacement algorithms based upon historical information alone or probabilistic information alone cannot always yield the best performance. One alternative is to make use of both of these sources of information in the design of new algorithms. Any such algorithm will still suffer from deficiency (i) cited in Section 1.1.2. Another alternative is to make use of structural information in place of probabilistic information, because of advantages pointed out earlier. The structural information we are referring to is the same as that adopted in the pagination schemes, i.e. page connectivity information. We shall, in particular, utilize information on the existence and lengths of paths between page-nodes.
We develop a replacement algorithm which uses structural information to make replacement decisions. The **Interval of Next Reference** (INR) replacement algorithm would replace a page that has the largest interval of first possible next reference from the presently referenced page. This algorithm is heuristically justifiable, since a good page to replace is the one which is expected to be referenced furthest away from the current page. If a page recently referenced has a very large value of INR from currently referenced page, then the earliest time that page can be referenced again is far away, hence the page is a good candidate for replacement. The INR replacement algorithm may be regarded as an approximation to Belady's optimal replacement strategy, which is unrealizable. INR replaces a page with highest lower bound to next reference, whereas Belady's scheme replaces a page with the highest value. The value of INR obtained from structural information guarantees that page will not be referenced before that interval. If the page replaced is one which has the largest value of INR, the "best" decision is made. In Section 3.2 we show that such an algorithm is optimal, for the assumptions made therein.

Next we consider a class of replacement algorithms which uses a mixture of historical and structural information. The use of structural information alone will quite often yield a number of pages which are suitable for replacement. Historical information is then used to break ties. The class of algorithms we consider here is called **Exemptive** replacement algorithms. These were first introduced by Lew [31]. These algorithms can be derived from the INR replacement algorithm. Exemptive algorithms use the historical information for
ranking pages in memory, and then use structural information to grant exemptions to certain pages. The degree of structural information used may vary. A simple subset of this class of replacement algorithms is one where (one-step) connectivity information among pages is used to grant exemptions. These algorithms prevent the catastrophe of a recently replaced page being referenced immediately, by exempting pages directly connected to the currently referenced page from replacement. A restricted sub-class can also be defined where only the last two historically ranked pages are candidates for replacement, and the non-exempt of the two is replaced. For Markovian assumptions on program execution we show later that such algorithms provide at least as good performance as corresponding historical replacement algorithms.

Another variation of these algorithms is where the "localities" determined during reordering and pagination are used to provide exemptions. The pages belonging to the current locality are exempted from replacement. This algorithm preserves the pages of the current locality in the main memory. In this case the replacement algorithm and the pagination procedures are mutually compatible.
1.4 MEMORY ALLOCATION

A third problem in virtual memory system management is that of memory allocation. The allocation problem arises because the limited main memory resources have to be shared dynamically among a number of users. A number of executing programs must be maintained in the system to optimize the use of the overall system resources and to provide high throughput. The memory allocation problem is comprised of the following subproblems:

I. Partitioning problem—determination of the amount of main memory to be allocated to each program.

II. Locality size estimation—determination of the amount of main memory required by a program for efficient execution.

III. Locality membership estimation—determination of the pages of a program which constitute the locality at any time.

The problem of deciding which programs should be allocated memory at all (i.e. be made executing) is a scheduling problem, which is outside the scope of this work.

1.4.1 Earlier Work

There have been two basic approaches to the solution of the first problem. The fixed partitioning approach divides the main memory into fixed sized partitions and constrains each user to remain within the allocated partition. The main problem then is that of determining the
partition size, or equivalently the optimal degree of multiprogramming. The dynamic partitioning approach places no constraint on the size allocated to the individual programs; there is of course a limit to the total amount of main memory available. In this case, partition size is dynamically varied according to the user's requirements.

Different programs have different needs and their needs vary with time. Thus any scheme which does not take the variations into account cannot be optimal. It has been shown, under various assumptions and for various performance criteria, that fixed storage partitioning is inferior to dynamic storage partitioning [32,55-59]. This is mainly because the dynamic scheme is more flexible and conforms to the individual needs of the program much better. If these needs can be quantitatively formulated as a function of partition size for each program, then mathematical optimization techniques are applicable [30]. Partitioning algorithms, however, need to be designed with care, otherwise "thrashing" [33] may result. To prevent thrashing, algorithms based upon the working set model of programs [8] have been designed. Essentially such algorithms attempt to allocate a partition size equal to the working set size, for each executing program. We recall that we regard working sets as dynamic estimates of localities (cf. Section 1.1.4).

The working set principle of memory management attempts to solve all three problems listed above simultaneously. The partition size to be allocated is made equal to the working set size and the membership of the working set is the set of pages referenced by the program in a certain interval of time immediately preceding the current time instant.
Various implementations [29,34-36,56] differ in their means of determining or estimating the working set dynamically. Rodriguez-Rosell and Dupuy [34] describe an implementation where the working set size is re-estimated every $\tau$ seconds (the window size). They establish the fact, through measurement, that by allocating memory equal to the working set size thrashing is indeed avoided. Chu and Opderbeck [24] consider a variation where $\tau$ is a variable, instead of being fixed. A page fault frequency threshold is maintained and more space is allocated if this value is exceeded. Otherwise, some space may be deallocated. The performance of this scheme is highly program dependent, e.g. if localities change very fast the program will keep on adding pages to its allocation, even though actually only a much smaller number of pages may be needed.

1.4.2 Present Work

Optimal Partitioning. Our objective here is to determine techniques of obtaining the optimal storage partitions, based upon the estimation of program requirements, and so as to optimize a cost criterion. The variable partitioning scheme that we discuss is one in which the optimal partitions are ascertained at various intervals (the execution intervals), and during these intervals the partition sizes are maintained constant. The execution interval may be fixed, e.g. equal to a 'time slice' or 'quantum', chosen according to processor scheduling criteria; or else it may be variable, e.g. equal to the time before an I/O 'block'. At the end of an interval, when a program comes up for reallocation its memory size may be readjusted, depending upon
the estimate of its requirements (the locality size), or it may be swapped out to the auxiliary memory. We shall consider the determination of optimal partitions using the Lagrange multiplier method of optimization.

**Estimation of Locality Size.** The second problem is the accurate estimation of locality sizes of programs. The working set estimate for locality size as defined by Denning [8], is the number of distinct pages referenced by the program during an interval $\tau$, in the immediate past. This definition, and most variations implemented, use historically measured information to estimate the locality size. Typical performance measures include page-fault rates, page residency times, page usage frequency, etc. The use of historical information alone is justifiable if programs satisfy the locality principle, i.e. if the localities change very slowly. However if the localities change faster, then historical information alone will not always provide a good estimate of locality size. The knowledge of a priori information about the mean and variance in size of the locality a program is currently executing would improve the estimate greatly. As we discussed earlier, the structural information about the program graph may be used to provide such a priori information. However, the use of structural information alone is not enough, because program execution is probabilistic and depends upon the history of program execution also. Thus a good method for estimation of locality size is one which makes a proper use of both historical and structural information. We recommend use of Bayesian Decision Theory [37] for this purpose, and we show through simulations that such estimates indeed lead to
considerable improvements in performance.

"Bayesian" approaches to memory allocation problems have been proposed before. In a general sense, any algorithm that utilizes observed information to make decisions adaptively is Bayesian. Shemer and Gupta have proposed such an adaptive scheme for memory allocation [38]. They base the allocation and replacement decisions on historical information and a priori probabilities obtained through trial runs. They give a memory allocation scheme somewhat similar to the working set scheme of Denning (they estimate working sets based upon referencing patterns of the program), but they fail to include considerations of thrashing or the use of a priori information (other than that dependent upon prior executions). We use a Bayesian approach in a more formal sense. Specifically, a Bayesian decision algorithm is one where a "decision" is sought, based upon a priori (structural) and sample (reference history) information, and that formally minimizes an expected loss. The distinguishing characteristic of our approach is the utilization of structural rather than solely historical information. The problem of thrashing is avoided through executing a program only if a sufficient amount of main memory resources is allocated to the program (this amount determined through optimal partitioning considerations).

Determination of Membership. The problem of determination of membership is that of deciding what pages to load as members of a locality, and what page(s) to unload when more pages are desired. This may be considered from the following points of view:

* Demand Paging.
* Pre-loading.
In demand paging the strategy used for loading pages is page faulting. The basic idea is to load only those pages which are actually needed. Once the partition allotted to a program is full, an appropriate replacement algorithm is used to decide which page to replace to make space for each demanded page. In pre-loading, the locality membership is estimated in advance and the entire locality is loaded together instead of one page at a time, as in demand paging. The idea is to save on the loading costs. In such an environment we recommend that pre-load sets be based upon the a priori estimates of localities obtained through structural considerations.

1.4.3 Implementation on BCC 500

The pre-loading algorithm considered by us is motivated by the design of the BCC 500 computer system. The BCC 500 is a time-shared virtual memory system where processes are given a quantum of CPU time after which they are unloaded to auxiliary memory. Before the next quantum the locality is estimated (using historical information) and pre-loaded in the main memory. We have simulated the implementation of a pre-loading scheme where the pre-load sets are based upon a priori locality information. At changes of localities (which can be detected by checking for a change in locality number upon page faults) a process blocks and pages belonging to the next locality are pre-loaded. Our simulation experiments show that such a scheme indeed leads to a more efficient operation of the system, yielding significant reductions in page-faulting. We also discuss implementation considerations and the feasibility of such a scheme with respect to the BCC 500 system.
CHAPTER 2
Reordering and Pagination

2.1 SEGREGATION OF DATA AND INSTRUCTION

We shall assume all code is re-entrant and hence do not propose mixing (integrating) instruction and data nodes onto the same page. An argument against this decision may be offered by considering the program given in Figure 2.1.1. In the figure \( \alpha \) denotes an instruction node and \( \beta \) denotes a data node. The label on a branch connecting two instruction nodes is the expected number of times that branch is executed, and the label on a branch connecting an instruction and a data node is the expected number of times the data node is referenced by the instruction node each time latter is executed. For a page size of two nodes, two possible paginations are shown in Figures 2.1.1 (b) and (c). For a resident set size of two (and optimal replacement) the two paginations produce \( 8nk + 2n \) and \( 8n \) faults, respectively. This points out that mixing data and instruction nodes onto the same page may be advantageous. However, when the resident set size is greater than 2 pages the situation is reversed. For a resident set size of 3 (and optimal replacement), the page faults produced by paginations (b) and (c) are \( 2n + 2 \) and \( 4n + 2 \), respectively. These two results are based upon the following node trace for the program:

\[
((\alpha_1 \beta_1 \alpha_1 \beta_3)^k \alpha_1 (\alpha_2 \beta_2 \alpha_2 \beta_4)^k \alpha_2 (\alpha_3 \beta_1 \alpha_3 \beta_3)^k \alpha_3 (\alpha_4 \beta_2 \alpha_4 \beta_4)^k \alpha_4)^n.
\]

We note that for a non-segregated pagination (Figure 2.1.1(c)) only two pages are required while each instruction node is being executed, whereas for segregated pagination (Figure 2.1.1(b)) three pages are required.
Figure 2.1.1

(a) Program node graph
(b) Segregated Pagination
(c) Non-Segregated Pagination

Figure 2.1.2

(a) Program Node Graph
(b) Segregated Pagination
(c) Non-Segregated Pagination
In the above example, we assumed that the data nodes are either referenced by a single instruction node or by instruction nodes which are close together. Consider an example where the same data node is referenced by instruction nodes which are structurally far apart, as in Figure 2.1.2. The page faults for a resident set size of two pages (and optimal replacement) are $8n$ and $12n$, respectively, for segregated and non-segregated paginations. For this case it is better to put instructions and data separately because the number of page faults for the second loop $\{a_5, a_6, a_7, a_8\}$ would increase otherwise. This example demonstrates that for programs where all data nodes are referenced by more than one instruction node, far apart structurally, segregated pagination may yield better performance, for a resident set size of $\geq 2$ pages.

Some factors which favor segregated pagination include:

- in general, instruction nodes refer to more than one data node, and the same data node may be referenced by several different (and structurally distant) instruction nodes.
- for re-entrant programs, it is imperative that segregated pagination be adopted (because distinct copies of data may be necessary).
- for segregated paginations, only the data needed by an instruction sub-locality need be loaded, thus improving the residency of both data and instruction pages.

Due to these factors, and due to the relative simplicity of the pagination schemes which result, we will adopt a segregated pagination policy.
2.2 REORDERING AND PAGINATION OF INSTRUCTION NODES

The criterion we adopt for deciding the best distribution of instruction nodes onto pages, as considered in section 1.2, is the minimization of space-time cost. The cost criterion (1.2.1) for instruction nodes may be stated as:

$$C_I = \int_0^{T_{\text{exec}}} x_I \quad \text{RS}_I(t) \, dt + \sum_{i=1}^{W} \text{RS}_I(t_i) t_i$$

(2.2.1)

where the subscript "I" indicates that the costs are for instruction pages alone. In particular, RS_I(t) is the memory space (resident set size) occupied by instructions at virtual-time t, and RS_I(t_i) at the i\textsuperscript{th} page fault.

We write

$$C_I = C^1_I + C^2_I$$

(2.2.2)

The minimization of cost \(C^1_I\) requires that the size of resident sets of instructions be small, and minimization of \(C^2_I\) requires in addition that the number of page faults be minimized. This implies that code should be paginated such that only those nodes which are actually needed are in memory at any time. We shall first discuss methods of reordering code to aid the pagination process.

2.2.1 Reordering

The objective of reordering is to put close together in space the code that will be required close together in time, so as to minimize
cost function (2.2.2). It is easy to see that it is not possible to achieve this for the entire program, so we try to keep together the portions with the greatest "nearness" [16] requirements. The instruction nodes with the greatest nearness requirement are those which belong to an SCR. According to the "locality" model of program behavior these are the constructs in the program graph which remain in memory during a major portion of the program's execution. Thus our main effort will be to determine the SCRs and to order the nodes of an SCR together.

The ordering we propose (with respect to an instruction node connectivity matrix) is what we call locality ordering, where the matrix elements corresponding to SCRs appear along the diagonal of the reordered matrix. This is obtained by permuting rows and columns of the matrix such that matrix elements corresponding to an SCR fall within a submatrix, and such that the submatrices corresponding to SCRs which communicate most heavily appear close together in the reordered matrix.

A locality ordering can be achieved through detection and merging of the nodes belonging to each SCR, and then use of topological sorting (Knuth [43]) on the reduced graph. This procedure keeps most of the non-zero elements close to the diagonal and only elements associated with transitions between SCRs appear away from the diagonal. The topological ordering produces an upper-triangular reordered matrix (if we assume all SCRs are merged into single nodes). Within an SCR, the ordering is again carried out by following the topological ordering procedure, but first we have to remove a branch ("break" the SCR) so that the partial ordering property required for topological ordering
is satisfied. The branch to be removed should be one such that no more
SCRs contained within the SCR being ordered are broken.

Prior to considering the procedure for ordering of instruction
tnodes, all the SCRs in the program instruction graph have to be located.
(The SCRs are also essential for the estimation of localities as
discussed later.) Numerous procedures for finding SCRs appear in the
literature. Aho et. al. give an efficient algorithm (Algorithm 5.4
[44]) for finding SCRs in a graph. The algorithm is of order Max(n,e)
for n nodes and e edges in a graph, thus overhead for locating SCRs is
proportional to this number. Note that for programs, some SCRs can be
located with very little overhead, e.g. the "DO loop" constructs, thus
reducing the overhead.

The ordering of instruction nodes is based upon the concept of
equally important nodes, which may be explained as follows:

• for the reduced graph (where all the SCRs are represented
  by single merged nodes), all nodes are equally important
  and may be ordered using topological ordering. Since there
  are no more loops, the partial ordering condition required
  is satisfied.

• for the merged nodes, all nodes contained therein are
  considered equally important and an ordering procedure
  similar to one above is followed.

In a topological sorting procedure, a slight ambiguity may arise.
Consider the graph shown in Figure 2.2.1. A topological sort would
order the nodes $\alpha_1$, $\alpha_2$, $\alpha_3$, $\alpha_4$, but then it can order either $\alpha_8$ or $\alpha_9$.
If $\alpha_9$ is ordered first, the nodes $\alpha_2$, $\alpha_3$, $\alpha_4$, and $\alpha_5$ will be separated
from $\alpha_8$ violating the nearness needs of these nodes. To prevent this
Figure 2.2.1 - A Prog. Example

(a) Reduced Graph

(b) Connectivity Matrix

Kernighan's Example

Figure 2.2.2

Figure 2.2.3
we modify the sorting procedure as follows: Prior to a node with more than one predecessor, order the nodes from which this node is reachable. Hence prior to $\alpha_5$, we order $\alpha_8$ before considering $\alpha_9$ and $\alpha_{10}$. Thus the ordering is $\alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_8 \alpha_5 \alpha_9 \alpha_{10} \alpha_6 \alpha_7$.

The reachability matrix produced by this ordering procedure satisfies the locality ordering considered earlier. The topological sort algorithm given below follows Knuth [43], except when an SCR is encountered all nodes of the SCR are ordered before proceeding. We first define the following:

- $i \preceq j$, if node $j$ is reachable from node $i$.
- LEVEL, the nesting level of SCR under consideration.
- NODE(LEVEL), present node under consideration.
- OPEN(LEVEL), nodes yet to be ordered.
- REACHABLE(LEVEL), stack of nodes reachable from nodes in OPEN(LEVEL) list.
- TOP(LEVEL), the exit node of PRESENT_GRAPH(LEVEL).
- PRESENT_GRAPH(LEVEL), the subgraph (SCR) being ordered presently.
- $|\text{PRESENT_GRAPH}(\text{LEVEL})|$, the number of instruction nodes in the SCR.
- ORDERED, list of ordered nodes.

The ordering algorithm is as follows:

a.1 Initialize: ORDERED = $\emptyset$, LEVEL = 1, REACHABLE(LEVEL) = $\emptyset$, OPEN(LEVEL) = $\emptyset$

PRESENT_GRAPH(LEVEL) = Reduced Graph.

a.2 Set TOP(LEVEL) = Exit node, NODE(LEVEL) = Initial node.
a.3 If \( \text{NODE(LEVEL)} \) is an SCR, then

Delete \( \text{NODE(LEVEL)} \) from \( \text{PRESENT\_GRAPH(LEVEL)} \).

Set \( \text{LEVEL} = \text{LEVEL} + 1 \).

Set \( \text{PRESENT\_GRAPH(LEVEL)} = \text{SCR corresponding to NODE(LEVEL-1)} \).

Delete an arc \((j, i)\) in \( \text{PRESENT\_GRAPH(LEVEL)} \), and

Set \( \text{NODE(LEVEL)} = i, \text{TOP(LEVEL)} = j \).

Repeat step a.3 onwards.

a.4 If \( \exists i \in \text{OPEN(LEVEL)} \ni \notin \text{NODE(LEVEL)} \), then

Stack \( \text{NODE(LEVEL)} \) on top of \( \text{REACHABLE(LEVEL)} \).

Set \( \text{OPEN(LEVEL)} = \text{OPEN(LEVEL)} - i \).

Set \( \text{NODE(LEVEL)} = i \).

Repeat step a.3 onwards.

a.5 Add \( \text{NODE(LEVEL)} \) to list of ordered nodes, i.e.,

Set \( \text{ORDERED(LEVEL)} = \text{ORDERED(LEVEL)} \cup \text{NODE(LEVEL)} \).

Delete \( \text{NODE(LEVEL)} \) from \( \text{PRESENT\_GRAPH(LEVEL)} \).

a.6 If \( \text{NODE(LEVEL)} = \text{TOP(LEVEL)} \), then set \( \text{LEVEL} = \text{LEVEL} - 1 \).

a.7 If \( \text{LEVEL} = 0 \), then Exit.

a.8 Set \( i \) = immediate successor of \( \text{NODE(LEVEL)} \).

If \( \text{NODE(LEVEL)} \) has more than one such successors, then

Choose \( i \in \{ i \in \text{REACHABLE(LEVEL)} | i = \text{REACHABLE(LEVEL)} \} \).

Add the remaining such nodes to the \( \text{OPEN(LEVEL)} \).

Set \( \text{NODE(LEVEL)} = i \).

If \( \text{REACHABLE(LEVEL)} = i \), then Unstack \( \text{REACHABLE(LEVEL)} \).

Repeat step a.3 onwards.

(A PL/I program for above algorithm is listed in Appendix E.)

We will next consider a few examples illustrating the above algorithm.)
Example 1. Kernighan's [11] example (Figure 2.2.4(a)).

The SCRs for the program graph are:

\[ \text{SCR}_{10} = \{5,6\}, \text{SCR}_{11} = \{7,8\}, \text{SCR}_{12} = \{2,3,4\}. \]

The ordering produced by the algorithm is,

1, 3, 4, 2, 6, 5, 8, 7, 9.

The reduced graph and the connectivity matrix are given in Figure 2.2.2.

Example 2. Baer and Caughey's [5] example (Figure 1.1.1).

The SCRs in the graph are:

\[ \text{SCR}_{25} = \{6,7\}, \text{SCR}_{26} = \{8,9\}, \text{SCR}_{27} = \{10,11\}, \]
\[ \text{SCR}_{28} = \{15,16\}, \text{SCR}_{29} = \{21,22,23\}, \text{SCR}_{30} = \{14,28,17\}, \]
\[ \text{SCR}_{31} = \{5,25,26,27,12\}, \text{SCR}_{32} = \{31,13,30,18\}. \]

The ordering produced by the above algorithm is,

\{1,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,3,20,21,22,23,24\}.

The reduced graph and the connectivity matrix are given in Figure 2.2.3.

Most non-zero elements are confined to region near the diagonal, and all nodes of an SCR are all ordered together as required for locality ordering.

2.2.2 Pagination

The pagination should be such as to preserve the ordering obtained as above while minimizing the cost (2.2.2). For minimizing \( C_{1}^{1} \), the resident set size should be kept low, and since the SCRs provide the estimates for resident sets the number of pages allocated to each SCR should be minimized.

Next consider the cost \( C_{1}^{2} \). Let the average resident set size be \( \bar{RS}_{1} \). Then
\[ C^2_I = \bar{R}_I \ t_w \ x_I \]
\[ = \bar{R}_I \ t_w \ \sum_{i=1}^{m} \sum_{j=1}^{m} \tau_{ij} \ p'_{ij} \]  
\[ (2.2.3) \]

where \( t_w \) = waiting time to load a page,
\( x_I \) = instruction page fault count,
\( \tau_{ij} \) = number of control transfers between nodes \( \alpha_i \) and \( \alpha_j \), and
\( p'_{ij} \) = probability that \( \alpha_i \) and \( \alpha_j \) are not in the same resident set.

In the literature, \( p'_{ij} \) is commonly assumed to be equal to one whenever
nodes \( \alpha_i \) and \( \alpha_j \) are not in the same page, equal to zero otherwise (which
is equivalent to assuming single page resident sets). Pagination is
then based upon assumed knowledge of \( \{ \tau_{ij} \} \). We shall instead consider
use of structural information for minimization of \( C^2_I \), which permits us
to drop the assumption of single page resident set, and define \( p'_{ij} \) more
realistically. We recall that "localities" provide natural estimates
for resident sets, because they are defined to be resident for much of
the time. Each locality also has a maximum communication among its own
nodes (thus minimizing the value of \( p'_{ij} \) for nodes belonging to a
resident set), and the least communication with other localities. Since
localities form the estimates for resident sets, and minimization of
costs \( C^1_I \) and \( C^2_I \) requires preserving these localities, we adopt the
following guidelines for pagination:

- SCR smaller than a page size should lie within a page.
- Each SCR (since the localities are based on them) should be contained in a minimum number of pages to minimize the cost \( C^1_I \).
- It is better to break up nodes than to leave big "holes" on pages. Presence of holes would increase the fragmen-
tation and spread code over a greater virtual address space. This would increase the number of pages required for the same resident set and hence result in a greater cost. The nodes may be split at any point since the nodes do not contain any loops (by definition).

- If SCR's or nodes are to cross page boundaries, the choice as to where these are to be broken must be such that:
  --within the SCR no more SCR's are broken.
  --if relative frequencies of execution are known, break the arc with minimum frequency.

Based on these ideas, we outline a pagination algorithm which uses the reordered connectivity matrix.

- Consider each MSCR and paginate nodes belonging to an MSCR as follows:
  - Find the minimum number of pages required by MSCR, i.e.,
    \[
    \left\lfloor \frac{\text{MSCR size}}{z} \right\rfloor
    \]
    * Within the MSCR allocate all 'DO loops' smaller than a page size to a page each. For nested loops and adjacent loops which fit within a page, allocate those to a page each.
    * Within the MSCR allocate pages to SCR's as follows:
      * Allocate a minimum number of pages, i.e., \( \left\lceil \frac{\text{SCR}}{z} \right\rceil \), to the nodes of an SCR (which are adjacent in the reordered matrix). Paginate such that holes appear only on the entry or exit pages of the SCR. (There may be more holes for SCR's within the SCR which are slightly smaller than a page size and are adjacent to one
another. Such SCRs may increase the total number of pages allocated to the MSCR over the minimum.)

• Repeat this for all SCRs within the MSCR. For SCRs for which part of the nodes have been allocated already, paginate such as to fill in holes left in entry/exit pages. The nodes may be split for this purpose.

• For the remaining nodes of the MSCR, allocate pages such as to fill in any holes on entry/exit pages of adjoining SCRs, within the MSCR. If no such possibility exists, then allocate nodes to pages such that adjoining nodes occur together.

• Repeat the above procedure for the remaining MSCRs of the program graph. After all MSCRs have been allocated to pages, the remaining nodes may be paginated as follows:

  • nodes which can fit onto immediate successor or predecessor pages may be allocated to those pages. (The nodes may be split.)

  • allocate the remaining nodes to pages in the order of their position in the reordered matrix.

(A PL/I program for above algorithm is listed in Appendix F.)

We will illustrate the above procedure using a few examples:

Example 1. (Kernighan's [11] example.) The program graph is shown in Figure 2.2.4(a). The paginations corresponding to Kernighan's procedure and our procedure are as follows (for a page size of 4):

<table>
<thead>
<tr>
<th>Kernighan's</th>
<th>Our's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>Nodes:</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
The page graphs corresponding to these paginations are shown in Figure 2.2.4(b) and 2.2.4(c). The page faults produced for various resident set sizes and LRU replacement are given in Table 2.2.1.

**Example 2.** (Baer and Caughey's [5] example.) The program graph is given in Figure 1.1.1. The paginations corresponding to Baer and Caughey's procedure and our procedure are as follows (for a page size of 8):

<table>
<thead>
<tr>
<th>Baer and Caughey's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</td>
</tr>
<tr>
<td>Nodes: 1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Our's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</td>
</tr>
<tr>
<td>Nodes: 5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

The page graphs corresponding to these paginations are shown in Figures 2.2.5(a) and 2.2.5(b).

For some reasonable values of the input variables (K1=1, I1, I2, I3=0), such that each section of the program is executed at least once, we generated the node trace shown in Figure 2.2.6. For this node trace, the page traces generated for above paginations are shown in Figure 2.2.7.

The page faults produced for these page traces for various values of resident set size, using the LRU replacement algorithm, are given in Table 2.2.2. Our pagination yielded general reductions in the total number of page faults, over the range of resident set sizes. This improvement is a consequence of the condition of minimizing the number
### Table 2.2.1

<table>
<thead>
<tr>
<th>Set Size</th>
<th>Kernighan</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

### Table 2.2.2

<table>
<thead>
<tr>
<th>Set Size</th>
<th>B &amp; C</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4955</td>
<td>4942</td>
</tr>
<tr>
<td>2</td>
<td>2284</td>
<td>2271</td>
</tr>
<tr>
<td>3</td>
<td>335</td>
<td>318</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>85</td>
</tr>
<tr>
<td>5</td>
<td>98</td>
<td>85</td>
</tr>
</tbody>
</table>

---

1 2 4 5 6 7 8 10 11 (12 \(T_1\)) \(4\) 12 13 14 15 \(16\) 16 \(17 \(T_2\)\) \(50\) \(18 \(T_1\)\)

\(12 \(T_1\)\) \(4\) 12 13 \(T_2\) \(17 \(T_2\)\) \(50\) \(12\) 18 19 20 21 22 (23 21 22) \(10\) 23 24

**Figure 2.2.6** - Node Traces

B&C: \(P_1P_2P_3P_4P_5P_6P_3T_1\) \(4\) \(P_7P_8P_9P_{10}P_9\) \(2\) \(P_{10}T_2\) \(50\) \(P_{10}T_1P_3T_1\) \(4\) \(P_3P_7T_2\)

\(P_{10}T_2\) \(50\) \(12\) \(P_{10}P_{11}P_{12}P_{13}P_{14}P_{13P_{14}}\) \(10\) \(P_{13}P_{12}\)

**Figure 2.2.7** - Page Traces

OUR: \(P_{11}P_{10}P_9P_{12}P_3P_{11}T_1\) \(4\) \(P_3P_9P_8P_5P_4P_{4P_5}\) \(2\) \(P_5T_2\) \(50\) \(P_{11}T_1T_2P_5T_2\) \(50\)

\(12\) \(P_9P_{12}P_7P_6P_7P_{11}\) \(10\)
of pages allotted to each SCR. For the page traces in question, the largest SCR, denoted by $T_2'$ and $T_2''$ in Figure 2.2.7 for two paginations, has more distinct pages for Baer and Caughey's pagination. By allotting the fewest possible pages to each SCR, less pages are referenced each time the SCR is referenced and this lowers the number of page faults.

**Example 3.** We consider another example which demonstrates clearly the advantage due to minimizing the number of pages in each SCR. Consider the program graph in Figure 2.2.8 with the execution frequencies as marked. The program is a predominantly "looping" program (e.g., a matrix multiplication program). The node sizes and the two paginations are as follows (assuming a page size of 8):

<table>
<thead>
<tr>
<th>Node number:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node size:</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Baer and Caughey's Pagination</th>
<th>Our Pagination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page: $P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$ $P_7$ $P_8$</td>
<td>$P_1$ $P_2$ $P_3$ $P_4$ $P_5$ $P_6$</td>
</tr>
<tr>
<td>Nodes: 1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

The page graphs for the two paginations are shown in Figure 2.2.9.

Due to the condition of minimizing the number of pages for SCRs, the two SCRs ($\{2,3,4,5\}$ and $\{1,2,3,4,5,6,7\}$) have a lesser number of pages in our pagination. (Some nodes cross page boundaries to achieve this.) The page faults produced by the two paginations, for the frequency of execution of nodes given earlier (Figure 2.2.8), are shown in Figure 2.2.10 drawn on a semi-log scale. As can be seen, our pagination produces a much smaller number of page faults for larger resident set sizes.
Figure 2.2.8 - Program Node Graph

Figure 2.2.9 - B & C's Page Graph

Figure 2.2.10 - Page Fault vs Res. Set Size

Figure 2.2.11 - Our Page Graph
As noted earlier, the data nodes deserve explicit treatment due to their different nature. The data nodes, for instance, may be referenced by instruction nodes belonging to different localities. Hence some data nodes may be referenced all over the program, while reference to instruction nodes is more localized. In consistency with these and earlier observations, we develop an ordering scheme for data nodes based upon the "localities" concept. The effort will be to minimize the space-time cost criterion considered earlier.

In the graphical representation of the program, the data nodes are treated as special nodes (cf. Section 1.1). The connection between instruction nodes and the data nodes is assumed to be through undirected arcs. The ordering and pagination is done according to the instruction localities referring to the data nodes. Then there are the following main requirements for pagination of data nodes:

- minimize the total number of page faults.
- generate smallest number of pages for data referenced by each instruction locality.
- increase the page residency, i.e. the average time spent in memory each time the page is loaded.

The first two requirements are direct consequences of the minimization requirement for the space-time cost, and the third follows from the first two.

We first distinguish between simple data (data nodes with size ≤ page size) and more complex data structures (matrices, arrays, lists, data bases). The latter may be regarded as a collection of simple data
structured in some fashion. Pagination of complex data involves splitting data nodes larger than a page size into smaller pieces, which in effect imposes another structure upon the constituent simple data. Pagination of simple data involves aggregating them into pages so that data needed together in time is put together in space. Such "need" may be based upon instruction usage rather than data structure information.

Reordering and pagination of matrices and arrays have been considered by a number of authors. McKeller and Coffman [45] have suggested storing matrices in a partitioned submatrix form and modifying algorithms accordingly. Moler [46] suggests changing the order of reference to matrix elements within loops to increase page residency for matrix pages. Elshoff [47] considers a number of factors for matrix pagination such as aligning reference patterns for matrix elements with storage mapping functions, altering algorithms to utilize page size information, and designing algorithms to take into account the working environment (e.g., replacement algorithm). An example of these considerations is given in Appendix A.

List and data base reordering has been considered by Yue and Wong [12], who base the reordering decision on the reference probabilities of the nodes of the structures. Patt [48] gives a method of representation of tree structures so as to minimize search time for elements. Bobrow and Murphy [49] discuss the structure of a LISP system for paged machines.

We restrict ourselves herein to the consideration of simple data (e.g., simple variables, small arrays, or nonstructured constituents of complex data). To paginate this data we consider the space-time cost criterion introduced in Section 1.2. The space-time cost, $C_D$, for data
nodes alone can be written as (from expression (1.2.1))

$$C_D = \int_0^{T_{\text{exec}}} R_S D(t) dt + \sum_{i=1}^{X_D} R_S D(t_i) t_w = C_D^1 + C_D^2 \quad (2.3.1)$$

where the subscript 'D' indicates that the costs are for data alone [cf. (2.2.1)]. $C_D^1$ is the cost during execution and $C_D^2$ is the cost due to page-faulting.

We will now derive expressions for the space-time cost, using an extension of the Markov chain formulation introduced by Lowe [10]. For the program graph $(I, D, E_I, E_D, \alpha_I, \alpha_D)$ defined in Section 1.1.3, we further define:

- $E = \text{instruction node initial entrance probability vector}$
- $P = \text{instruction node transition probability matrix}$
- $z = \text{page size}$

$E$ is a row vector $(e_1, \ldots, e_m)$.

A pagination of $I = \{\alpha_1, \ldots, \alpha_m\}$ and $D = \{\beta_1, \ldots, \beta_n\}$ may be described as a partition:

$$\{A_1, A_2, \ldots, A_w\} \ni i \neq j \Rightarrow A_i \cap A_j = \emptyset, \text{ and } \bigcup_{i=1}^{w} A_i = I$$

$$\{B_1, B_2, \ldots, B_K\} \ni u \neq v \Rightarrow B_u \cap B_v = \emptyset, \text{ and } \bigcup_{u=1}^{K} B_u = D \quad (2.3.2)$$

with the size of each $A_i$ and $B_u \leq z$.

For a given pagination, let $R$ and $S$ be boolean matrices such that:

$$r_{ij} = 1 \iff (\exists A_w) \ni (\alpha_i, \alpha_j \in A_w)$$

$$s_{uv} = 1 \iff (\exists B_k) \ni (\beta_u, \beta_v \in B_k)$$
Matrices \( R \) and \( S \) may be said to define paginations of instruction and data nodes, respectively. The transfer of control among instruction nodes can be represented by an absorbing Markov chain, if we introduce \( \alpha_{m+1} \) as the absorbing state. Then

\[
F = [I - P]^{-1} = \sum_{j=0}^{\infty} P^j
\]

where \( f_{ij} = \{ \text{expected number of times instruction node } \alpha_j \text{ is executed (before absorption)} \mid \text{initial node is } \alpha_i \} \)

Define \( \gamma_j = \text{expected number of times } \alpha_j \text{ is executed} \)

\[
= \sum_{i=1}^{m} e_i f_{ij}
\]

and \( \tau_{ij} = \text{expected number of control transfers between } \alpha_i \text{ and } \alpha_j \)

\[
= \gamma_i p_{ij}
\]

\( T_{\text{exec}} \) is the expected time of absorption.

We should mention here that the algorithms developed for pagination make use of relative values of \( \gamma_j \)'s for the various SCR's in the program graph and the absolute values are not required. The absolute values of \( \gamma_j \) would depend upon accurate knowledge of the \( P \) matrix, which as we discussed earlier may be difficult to obtain realistically.

2.3.1 Page Fault Analysis for Data Nodes

Let \( \{ \Delta(c) \} \) describe a sequence of instruction node executions before absorption, i.e.

each \( \Delta(c) = \alpha_i \in I, \text{ for } c = 1, 2, \ldots, f, \)

where \( f = \text{the total number of instruction nodes executed in time, } T_{\text{exec}}. \)

Let \( \sigma(\alpha_i) = \{ \delta_k \mid \exists u \in Q_i \quad (q_{iu} = 1 \land \beta_u \in B_k) \} \),
the set of data pages containing data nodes referenced by \( \alpha_j \). The number of references to a data node, which does not belong to any data page referenced by the previous instruction node, is given by

\[
|\sigma(\Delta(1))| \quad \text{for} \quad c = 1,
\]

\[
|\sigma(\Delta(c)) \cap \overline{\sigma(\Delta(c-1))}| \quad \text{for} \quad c = 2, 3, \ldots, f.
\]

If we assume that the first instruction node referenced is a dummy entry node which does not reference any data nodes, the total number of "new" data pages referenced by an instruction sequence \( \{\Delta(c)\} \) is given by

\[
\sum_{c=2}^{f} \left| \sigma(\Delta(c)) \cap \overline{\sigma(\Delta(c-1))} \right|
\]

The expected number of data page faults (assuming the memory comprises data pages referenced by the current instruction node), is the above sum weighted by instruction node transition probabilities,

\[
\chi_D = \sum_{c=2}^{f} \left\{ \sum_{i=1}^{m} \sum_{j=1}^{m} \Pr[\Delta(c) = \alpha_j \land \Delta(c-1) = \alpha_i] \cdot |\sigma(\alpha_j) \cap \overline{\sigma(\alpha_i)}| \right\}.
\]

Whenever instruction node \( \alpha_j \) is executed, it references the same set of data nodes, the size of this set being given by \( \sum_{u=1}^{n} q_{ju} \). Hence

\[
|\sigma(\alpha_j)| = \sum_{u=1}^{n} \{q_{ju}(1 - \delta_{\alpha_ju})\},
\]

where

\[
\delta_{\alpha_ju} = \begin{cases} 
1, & \text{if } \sum_{v=1}^{u-1} q_{juv} \geq 1, \text{ i.e. if data node } \beta_u \text{ is in the same data page as a data node } \beta_v \text{ referenced by } \alpha_j, \text{ with a smaller subscript } v < u \\
0, & \text{otherwise}
\end{cases}
\]
Let
\[
\psi_{\alpha_j u} = \begin{cases} 
1, & \text{if } \sum_{v=1}^{n} q_{jv} s_{uv} \geq 1, \text{ i.e. if data node } \beta_u \text{ is contained in a data page referenced by } \alpha_j \\
0, & \text{otherwise}
\end{cases}
\]

Then
\[
|\sigma(\alpha_j) \cap \overline{\sigma(\alpha_i)}| = \text{number of distinct data pages referenced by } \alpha_j, \text{ not common to data pages referenced by } \alpha_i
\]
\[
= \sum_{u=1}^{n} q_{ju} (1 - \delta_{\alpha_j u}) (1 - \psi_{\alpha_i u})
\]

Also, since
\[
\tau_{ij} = \sum_{c=2}^{f} \Pr [\Delta(c) = \alpha_j \land \Delta(c-1) = \alpha_i]
\]

Therefore
\[
x_D = \sum_{i=1}^{m} \sum_{j=1}^{m} \tau_{ij} \left\{ \sum_{u=1}^{n} q_{ju} (1 - \delta_{\alpha_j u}) (1 - \psi_{\alpha_i u}) \right\} \tag{2.3.3}
\]

In the above development it was assumed that only data pages referenced by the current instruction node are kept in the memory. But in practical memory management algorithms a "locality" of pages is maintained in the main memory. (We defer discussion on estimation of localities; for the present we define localities to be the set of pages most likely to be needed over an interval of time.) If all pages required by the current locality are maintained in the main memory, the expected number of data page faults is given by
\[
x_D = \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} \left\{ \sum_{u=1}^{n} q_{ju} (1 - \delta_{\alpha_j u}) (1 - \psi_{\alpha_i u}) \right\} \tag{2.3.4}
\]
where \( \tau_{ij} \), \( q_{ju} \), \( \delta_{ij} \), \( u \), \( \psi_{ij} \) are now defined in a similar fashion as before, but for instruction localities instead of for instruction nodes, \( r \) is the total number of localities for the program, and \( \ell_i \) and \( \ell_j \) are the \( i \text{th} \) and \( j \text{th} \) localities respectively.

The \( Q \) matrix defined for instruction localities is an \( r \times n \) matrix, where each row gives the data nodes referenced by the corresponding locality. Assume a data pagination, \( B \), and reorder the data nodes (columns of \( Q \)) so that data nodes belonging to a data page occur together. Let there be \( K \) data pages. Then define a boolean \( r \times K \) matrix \( Q' \) as:

\[
q'_{ij} = \begin{cases} 
1, & \text{if locality } \ell_i \text{ refers to a data node in } j \text{th data page} \\
0, & \text{otherwise}
\end{cases}
\]

The expected number of data page faults is then given by

\[
\xi_D = \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} \cdot \left\{ \text{number of non-zero elements of } j \text{th row of } Q' \text{ for which the entry in corresponding column of } i \text{th row is zero} \right\}
\]

\[
= \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} \cdot |Q'_{ij} \land \overline{Q_{ij}}|
\]

(2.3.5)

where \( Q'_{ij} \) is the \( j \text{th} \) row of boolean matrix \( Q' \).

\( |V| \) is defined as the number of non-zero elements in boolean vector \( V \).

We can now write expressions for space-time costs (2.3.1) for execution and data page faults as

\[
C_{D}^{1} = \int_{0}^{T_{exec}} \left( R_{S_{D}}(t) \right) dt = \sum_{i=1}^{r} y_{i} t_{\ell_i} |Q'_{i}|
\]

\[
= \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q'_{ij}| t_{\ell_i}
\]
where $t_{\ell_1}$ = average execution time spent in $\ell_1$, every time locality $\ell_1$ is entered.

$$C_D^2 = \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} \overline{\text{RS}}_{Dij} |Q_i'\cap Q_j'| \cdot t_w$$

where $\overline{\text{RS}}_{Dij}$ = the average memory space occupied by data pages during transition from $\ell_1$ to $\ell_j$.

$$= |Q_i'|.$$ 

Therefore,

$$C_D^2 = \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q_i'| |Q_j'| \cdot t_w$$

The overall space-time cost, $C_D$, is then given by

$$C_D = \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q_i'| t_{\ell_1} + \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q_i'| |Q_j'| \cdot t_w$$

$$= \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q_i'| \left[ t_{\ell_1} / t_w + |Q_j'| \cdot \overline{Q_i'} \right] t_w \quad (2.3.6)$$

The minimum space-time cost, minimal over all possible data paginations, $B$, is given by

$$C_D^* = t_w \left[ \min_B \left\{ \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q_i'| \left( t_{\ell_1} / t_w + |Q_j'| \cdot \overline{Q_i'} \right) \right\} \right]$$

(2.3.7)

This cost depends upon average wait time ($t_w$), page size ($z$), and average instruction locality execution times ($t_{\ell_1}$). The dependence of cost on page size $z$ is not explicit, but implicit, hence it complicates solution of (2.3.7). Also $|Q_i'|$ and $|Q_j'| \cdot \overline{Q_i'}$ are interdependent,
further complicating matters. Thus while an analytic solution to (2.3.7) cannot be obtained in general, enumerative methods can be used to obtain the optimal solution.

It is not possible to use a stepwise implicit enumeration technique, e.g. the backtrack programming algorithm in [10], because it requires at least weak monotonicity of the cost $C_D$, as a function of the number of data nodes. It can be shown by a counterexample (see Appendix B) that the cost $C_D$ does not satisfy this criterion. Thus to arrive at the optimal pagination, exhaustive enumeration of all possible paginations appear necessary without additional assumptions. The number of such paginations, which have to be tried before arriving at the optimal one, is extremely large, of the order $n!$, where $n$ is the total number of data nodes. Hence the method is impractical. However, if desired, an enumerative search procedure for locating the optimum can be based upon the algorithm of Johnson [50] which generates all possible permutations of the data nodes.

2.3.2 Reordering Procedures

In practice, it will be quite impractical to consider all possible paginations, due to the exhorbitant cost of such a scheme. Hence for practical pagination schemes, we consider a number of heuristic suboptimal solutions. In the expression for minimum space-time cost (2.3.7), the two factors which are dependent on the data pagination, $B$, and need to be minimized, are:

a) the number of data pages referenced by instruction locality $l_1$, and
b) the number of data pages referenced by \( l_j \) not common to data pages referenced by \( l_1 \).

The task of finding a sub-optimal solution is complicated by the fact that these two conditions are inter-dependent and hence cannot be independently minimized. Thus the heuristic solutions that we shall offer, would either favor minimization of (a) or minimization of (b). The relative importance of (a) or (b) depends upon the ratio \( t_{l_1} / t_w \).

We consider two cases:

**Case I.** \( t_{l_1} / t_w \) is of the same order as \(|Q_j'|, \, \overline{Q_1'}| \), or smaller.

In this case, the entire expression (2.3.7) is of importance and both factors (a) and (b) need to be minimized. We thus propose Heuristic I, which tries to achieve the objective by minimizing \(|Q_j'|, \, \overline{Q_1'}| \) for the largest values of \( \tau_{ij} \) in the \([\tau]\) matrix. The general procedure is to:

* order the data nodes for instruction localities corresponding to the largest \( \tau_{ij} \).
* minimize the data pages referenced by \( l_j \) which are not common to data pages referenced by \( l_1 \).
* minimize data pages referenced by \( l_1 \).

An outline for the ordering and pagination procedure follows:

b.1 Minimize \(|Q_j'|\) for locality \( l_j \) corresponding to the largest \( \tau_{ij} \), for which there remain some data (possibly also referenced by other localities), yet to be paginated. Paginate these data nodes on to a minimum number of data pages as follows:

b.1.1 Minimize \(|Q_j' \wedge \overline{Q_1'}|\) by spreading common data between \( l_1 \) and \( l_j \) as follows:
b.1.1.1 Spread the common data equally among the pages of $|Q'_j|$.

b.1.2 For localities corresponding to decreasing values of $\tau_{ku}$, such that $k,u \notin \{i,j\}$, until all data nodes have been paginated, do the following:

b.1.2.1 Minimize $|Q'_u|$ for $\ell_u$.

b.1.2.2 Minimize $|Q'_u| \wedge \overline{|Q'_k|}$ according to b.1.1.1.

This is done such that $|Q'_j| \wedge \overline{|Q'_k|}$ and $|Q'_i|$ are not increased for higher $\tau_{ij}$'s.

b.2 Repeat b.1 for remaining data nodes, until all data nodes have been paginated.

(A PL/I program for this pagination algorithm is listed in Appendix G.)

Case II. $t_{\ell_i}/t_w \gg |Q'_j| \wedge \overline{|Q'_k|}$ for all $i,j$.

In this case the expression (2.3.7) may be approximated as follows:

$$C^*_D = t_w \left[ \min_B \left\{ \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} \left| Q'_i \right| / t_{\ell_i} \right\} \right]$$

$$= \min_B \left\{ \sum_{i=1}^{r} \left| Q'_i \right| \gamma_{ij} t_{\ell_i} \right\}$$

(2.3.8)

since $\gamma_{ij} = \sum_{j} \gamma_{ij}$.

For this then, minimization of factor (a) alone is of importance, i.e. we need to minimize the total number of data pages for the instruction localities which have larger $\gamma_{ij} t_{\ell_i}$ products. We base a second heuristic procedure on this observation: Heuristic II minimizes the number of data pages connected to the localities in the order of their importance (according to values of the products $\gamma_{ij} t_{\ell_i}$). The general procedure is to:
• Order the instruction localities according to the decreasing values of $\gamma_j t_{\lambda_j}$.
• Minimize $|Q'_{ij}|$, the number of data pages referenced by $\lambda_i$, in decreasing order of priority of the $\lambda_j$'s.

A more detailed description follows later in this section.

We first establish that the asymptotic behavior, with respect to $t_{\lambda_1}/t_w$, of Heuristic II is better than that of Heuristic I under the assumption that $t_{\lambda^*}$ is the same for all localities. This is reasonable because all localities are chosen such that the program executes within the locality for some fixed period of time, $t_{\lambda^*}$, before changing to a new locality. Thus we assume the following:

$$\frac{t_{\lambda_1}}{t_w} = \frac{t_{\lambda_2}}{t_w} = \ldots = \frac{t_{\lambda_r}}{t_w}$$

(2.3.9)

Then it is easy to see that the cost function, $C_D$, is a linear function of this ratio. The expression (2.3.6) may now be rewritten as:

$$C_D = \bar{x} t_w \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q'_{ij}| + t_w \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q'_{ij}| - |Q'_{ij} - \bar{Q'}_{ij}|$$

Differentiating with respect to $\bar{x}$, for obtaining the slope of cost curve,

$$\frac{dC_D}{d\bar{x}} = t_w \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q'_{ij}|$$

(2.3.10)

Expression (2.3.10) gives the slope of cost curve with respect to $\bar{x}$, under assumption (2.3.9). The pagination which minimizes this slope can be found as follows:

$$\min_B \left\{ \frac{dC_D}{d\bar{x}} \right\} = t_w \min_B \left\{ \sum_{i=1}^{r} \sum_{j=1}^{r} \tau_{ij} |Q'_{ij}| \right\}$$

(2.3.11)

Since Heuristic II minimizes the factor on the right hand side of
expression (2.3.11) the slope of the cost curve for Heuristic II, with respect to $\bar{x}$, is smaller than that for Heuristic I. Hence Heuristic II will exhibit better cost performance with increasing value of $\bar{x}$. Also since the cost $C_D$ is a linear function of variable $\bar{x}$, there exists a threshold value of $\bar{x}$, above which Heuristic II yields lower cost as compared to Heuristic I (unless both heuristics yield an identical pagination for some example). The value of this threshold is not constant (and may be negative), but changes from one case to another. But in general if a large enough value of $\bar{x}$ is selected (by choosing localities accordingly), Heuristic II will yield a better data pagination.

We consider below two examples of 5 instruction localities and many more data nodes. The inter-locality frequency matrices ($\tau$) used for reordering the data nodes and for calculating costs of paginations, are given in Figure 2.3.1. (We assume the same value of $t_{\text{eq}}$ for all localities.)

**Example 1.**

The unordered data node reference matrix $Q$:  

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

(Here an "x" in row i, column j indicates that locality $\ell_1$ references data node j.)
Inter-Locality Reference Frequency Matrices

**Figure 2.3.1**

**Table 2.3.1**

Space-time Costs for Example 1

<table>
<thead>
<tr>
<th></th>
<th>Page Size = 4</th>
<th></th>
<th>Page Size = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Scheme</td>
<td>I</td>
<td>II</td>
<td>I</td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>633</td>
<td>624</td>
<td>804</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>418</td>
<td>534</td>
<td>556</td>
</tr>
<tr>
<td>(\tau_3)</td>
<td>542</td>
<td>546</td>
<td>703</td>
</tr>
</tbody>
</table>

**Table 2.3.2**

Space-time Costs for Example 2

<table>
<thead>
<tr>
<th></th>
<th>0.5</th>
<th>1.0</th>
<th>5.0</th>
<th>10.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>II</td>
<td>I</td>
<td>II</td>
</tr>
<tr>
<td>Scheme</td>
<td>I</td>
<td>II</td>
<td>I</td>
<td>II</td>
</tr>
<tr>
<td>(\tau_1)</td>
<td>371</td>
<td>430</td>
<td>534</td>
<td>568</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>350</td>
<td>392</td>
<td>472</td>
<td>504</td>
</tr>
<tr>
<td>(\tau_3)</td>
<td>354</td>
<td>437</td>
<td>487</td>
<td>561</td>
</tr>
</tbody>
</table>
The reordered data node matrix for $\tau_1$, using Heuristic I.

\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & 2 & 3 & 2 & 1 \\
7 & 9 & 3 & 1 & 5 & 7 & 1 & 9 \\
\end{array}
\]
\[
\begin{array}{cccccccc}
1 & 1 & 2 & 3 & 1 & 1 & 2 & 1 \\
5 & 1 & 5 & 2 & 0 & 4 & 6 & 8 \\
\end{array}
\]

The reordered data node matrix for $\tau_1$, using Heuristic II.

\[
\begin{array}{cccccccc}
2 & 2 & 1 & 1 & 3 & 2 & 2 & 3 \\
4 & 0 & 8 & 2 & 4 & 0 & 2 & 6 \\
\end{array}
\]
\[
\begin{array}{cccccccc}
1 & 1 & 2 & 2 & 1 & 2 & 1 & 2 \\
9 & 1 & 7 & 5 & 9 & 1 & 7 & 9 \\
\end{array}
\]

The space-time costs for various values of $\overline{x}$ are shown in Table 2.3.1.

(All costs are multiples of $t_w$, the waiting time.)

Example 2.

The data node matrix $Q$:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]
\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 9 \\
\end{array}
\]

The reordered data node matrix for $\tau_1$, using Heuristic I.

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 \\
2 & 4 & 8 & 6 & 0 & 1 & 4 & 2 \\
\end{array}
\]
\[
\begin{array}{cccccccc}
1 & 1 & 2 & 1 & 1 & 1 & 1 & 1 \\
5 & 1 & 9 & 4 & 7 & 0 & 3 & 6 \\
\end{array}
\]
The reordered data node matrix for $\tau_1$, using Heuristic II.

\[
\begin{array}{cccccccccccc}
1 & 1 & 2 & 1 & 1 & 1 & 2 & 2 & 1 & 1 & 2 & 2 & 1 & 1 \\
5 & 8 & 0 & 3 & 7 & 8 & 5 & 1 & 4 & 2 & 9 & 4 & 6 & 1 & 2 & 0 & 1 & 7 & 3 & 4 & 6 & 2 & 3 & 9 \\
\end{array}
\]

The space-time costs for a page size of 4 are given in Table 2.3.2.

For these examples the cost of Heuristic II increases less rapidly as compared to the cost of Heuristic I, as expected. In all the examples considered here there is a positive threshold value of less than 10.

In the discussion so far we have assumed that the instruction localities are known a priori. Denning's working set estimate of localities comprises the information referenced within last $\tau$ time interval, and depends only upon the history of program execution. There are two problems in estimating localities in this manner, namely

- we need a sample execution of the program to determine the various localities, and the execution times $t_{j1}$.
- the number of such localities may be too large so as to make the above solution procedure impractical to apply.

Hence some a priori estimates of the localities are needed. If we assume the locality model, where localities are the sets of pages (information) in which the program spends major portions of its execution time, then estimates of the localities in a program can be based upon structural information. (A procedure for determining localities is explained in Section 2.4.) In particular,SCRs may be used to obtain
a priori estimates of localities. Since we do not want localities to change very rapidly (i.e. to have small $t_{ji}$), and also we want to limit the total number of localities (otherwise the a priori information required to be maintained for memory management, as well as the data pagination algorithm time, increases), we limit our choice of SCR-based instruction localities to those whose average execution time is relatively large. As shown earlier, Heuristic II would then be a better scheme for data pagination.

We now give an algorithm for pagination using Heuristic II. For the algorithm we need to know the relative values of the products $Y_{ji} t_{ji}$ to determine a relative ordering of the localities. We discuss how these products can be determined in Section 2.4. Assume that the instruction localities $\{l_1, l_2, \ldots, l_r\}$ are priority-ordered, so that

$$i < j \text{ for } l_i, l_j, \text{ if } Y_i t_{l_i} \geq Y_j t_{l_j}; \quad (2.3.12)$$

$l_1$ is said to have highest priority.

Define $S_i = \text{the set of localities which reference data node } \beta_i, \beta_i \in D.$ Then for data node reordering, the following underlying principle is employed:

"Reorder the data nodes such that the sets of localities, $S_{l_1}$ and $S_{l_2}$, referring to consecutive data nodes $\beta_{l_1}$ and $\beta_{l_2}$, of the reordered set, differ minimally, i.e. either $S_{l_1} = S_{l_2}$ or they differ by a set X, such that the localities in X have lowest possible priority."

Based on this principle we propose the following reordering procedure:

R.1 Choose as the first reordered node a data node referenced by

locality $l_1$ and by a minimum number of other localities (of lowest
priority).

Let \( S_1 \) = set of localities which reference this data node.

\[ S_1 = \{ l_1 \} \text{ if only } l_1 \text{ references the data node.} \]

R.2 Order the data node referenced by the members of the set \( S_2 \) of localities, where \( S_2 \) is such that

\[ S_1 \cap S_2 \cap X = \emptyset \quad (2.3.13) \]

and set \( X \) is such that following conditions are satisfied:

i) \( X = S_1 \cup S_2 - S_1 \cap S_2 \)

ii) \( X \) is such that \( j_1 \) is largest among remaining data nodes, where

\[ j_1 \geq j \quad (\forall j \quad \exists \ l_j \in X. \quad (2.3.14)) \]

iii) In case more than one data node satisfies condition (ii), choose the data node which yields a minimum value of \(|X|\).

If a tie persists, choose a data node for which \( j_2 \) is largest, where

\[ j_2 \geq j \quad (\forall j \neq j_1) \quad \exists \ l_j \in X. \quad (2.3.15) \]

R.3 Let \( S_1 = S_2 \) and repeat step R.2 till all data nodes referenced by one or more localities has been ordered.

(A PL/I program for the above pagination algorithm is listed in Appendix H.)

Consider the earlier example of 5 localities and 32 data nodes (Example 1), where we assume the localities are priority-ordered. Then the above procedure produces the following reordered set of data nodes,
The reordered Q matrix is as follows:

\[ \begin{array}{ccccccc}
1 & 2 & 2 & 2 & 2 & 3 & 1 \\
7 & 5 & 9 & 1 & 5 & 7 & 3 \\
1 & 2 & 1 & 1 & 3 & 2 & 2 \\
2 & 2 & 1 & 2 & 1 & 0 & 4 \\
3 & 2 & 6 & 4 & 0 & 2 & 6 \\
4 & 0 & 4 & 0 & 2 & 8 & 4 \\
5 & 6 & 0 & 4 & 0 & 2 & 8 \\
\end{array} \]

In the reordered matrix all data nodes referenced by locality \( l_1 \) are ordered together, in a sequence. In the reordering procedure, step R.2, conditions (ii) and (iii) insure that all data nodes referenced by the highest priority locality are ordered sequentially. This is because by condition (ii) the difference in the sets of localities referring to consecutive data nodes, in the reordered set of data nodes, will be localities of lowest possible priority. This allows the maximum number of data nodes, referenced by the same locality, to be placed together, within the constraint that for higher priority localities also referring to the same data nodes, the ordering remains sequential.

For the above reordering procedure it can be seen that data nodes referenced by locality \( l_1 \) and those referenced by \( l_1 \) and \( l_{i-1} \) occur together (in a sequence) in the reordered set of data nodes, within the restriction that they all are referenced by the same set of localities of higher priority (i.e. priority higher than that of \( l_{i-1} \)). Thus for example, the data nodes referenced by \( l_3 \) and \( l_1 \), and data nodes referenced by \( l_3 \), \( l_2 \), and \( l_1 \), i.e. the data nodes \( \beta_{13}, \beta_{29}, \beta_{21}, \beta_5, \)
\(\beta_7, \beta_{23}, \beta_{31}, \beta_{15}\), in the above example, are all ordered together in the reordered set.

It can be shown that data nodes \(\beta_{i_1}, \beta_{i_2}, \ldots, \beta_{i_k}\), referenced by \(l_{i-1}\) or \(l_i\) and \(l_{i-1}\), occur together in the reordered set of data nodes, where \(\beta_{i_1}, \beta_{i_2}, \ldots, \beta_{i_k}\) are also referenced by the members of a set \(S\) (maybe \(\emptyset\)), such that

\[
j < (i-1) (\forall j) \exists \ell_j \in S\tag{2.3.16}
\]

i.e. all localities of the set \(S\) are of higher priority than \(l_{i-1}\). Let the reordered sequence of these data nodes be \(\beta_{j_{i+1}}, \beta_{j_{i+2}}, \ldots, \beta_{j_{i+k}}\). We will show this by contradiction.

Consider three nodes which occur in sequence in the reordered set. Let these data nodes be \(\beta_{j_{i+u}}, \beta_{j_{i+u+1}}, \beta_{j_{i+u+2}}\). We will show that if \(\beta_{j_{i+u}}\) and \(\beta_{j_{i+u+2}}\) are referenced by \(l_i\) (and set \(S\)), then \(\beta_{j_{i+u+1}}\) also has to be referenced by \(l_i\). Consider the following two cases:

**Case I.** Let \(\beta_{j_{i+u}}\) and \(\beta_{j_{i+u+2}}\) be referenced by members of the set \(S_1\).

\[S_1 = \{l_1\} \cup S,\]

where \(S\) is defined by (2.3.16). Let \(\beta_{j_{i+u+1}}\) be referenced by members of set \(S_2\), where

\[S_2 = S.\]

Then \(\beta_{j_{i+u}}, \beta_{j_{i+u+1}}, \text{and} \beta_{j_{i+u+2}}\) cannot occur in a sequence in the reordered set because it violates condition (ii). At the time of reordering of \(\beta_{j_{i+u+1}}\), there existed a node, namely \(\beta_{j_{i+u+2}}\), which had \(j_1(2.3.14)\) greater than the corresponding \(j_1\) for \(\beta_{j_{i+u+1}}\).
**Case II.** Let the data nodes \( j_{i+u}, j_{i+u+1}, j_{i+u+2} \) be referenced by members of the sets \( S_1, S_2, S_3 \), respectively, where

\[
S_1 = \{ \ell_i \} \cup S,
S_2 = \{ \ell_{i-1} \} \cup S,
S_3 = \{ \ell_i \} \cup \{ \ell_{i-1} \} \cup S.
\]

Again these data nodes cannot occur in a sequence in the reordered set, because this violates condition (iii). The value of \( |X| \) for \( \beta_{j_{i+u+1}} \) is greater than that for \( \beta_{j_{i+u+2}} \), hence \( \beta_{j_{i+u+2}} \) should be reordered before the node \( \beta_{j_{i+u+1}} \).

Hence \( \beta_{j_{i+u+1}} \) has to be referenced by \( \ell_i \) to occur adjacent to nodes \( \beta_{j_i+u} \) and \( \beta_{j_{i+u+2}} \) in the reordered set. This will be true for nodes referenced by all localities. Thus all nodes referenced by \( \ell_i \) and also referenced by members of set \( S \) (2.3.16) are ordered together by the reordering procedure.

**2.3.3 Pagination Procedure**

For the above procedure of reordering data nodes, we recommend a simple pagination procedure which minimizes \( |Q_1'| \) in order of locality priority. Assume localities are priority-ordered, data nodes are equal in size, and the page size \( z \) is in data node units. The reordering procedure reorders all the data nodes referenced by \( \ell_i \) together. The simplest way to paginate is to assign the first \( z \) nodes to page \( B_1 \), the next \( z \) nodes to page \( B_2 \), and so on. It is easy to see that this minimizes \( |Q_1'| \). For \( \ell_2 \), all nodes will be ordered together, hence the upper bound on the number of pages for these data nodes is,
The data nodes referenced by \( l_3 \) will be reordered into two groups, one also referenced by \( l_1 \), the second not referenced by \( l_1 \). Thus the maximum number of pages for data nodes referenced by \( l_3 \) are

\[
|Q'_3| \leq \left\lceil \frac{|Q_3|}{z} \right\rceil + 2
\]

For data nodes referenced by \( l_1 \), there will be at most \( 2^{i-2} \) groups. The data nodes in each group occur together in the reordered set of data nodes. Thus the maximum number of pages to which data nodes referenced by \( l_1 \) may belong is given by

\[
|Q'_1| \leq \left\lceil \frac{|Q_1|}{z} \right\rceil + 2^{i-2}, \quad i \geq 2
\]

where

\[
\left\lceil \frac{|Q_1|}{z} \right\rceil = \text{minimum number of pages that have to be assigned to data nodes referenced by locality } l_1.
\]

Thus if the reordered set of data nodes is \( \beta_{r_1}^{r_1}, \beta_{r_2}^{r_2}, \ldots, \beta_{r_n}^{r_n} \), the data pagination is given as follows:

\[
B_1 = \{ \beta_{r_z(i-1)+1}^{r_z(i-1)+2}, \ldots, \beta_{r_zi}^{r_zi} \}, \quad 1 \leq i \leq K \quad (2.3.17)
\]

This satisfies condition (2.3.2).

We will next consider the example of B & C (Baer and Caughey [5]), again, to demonstrate the effectiveness of the reordering scheme proposed here. The reordered data node matrix is (for simple data):
where \( \xi_1 = \{14,15,16,17\} \), \( \xi_2 = \{21,22,23\} \), and 'rest' is the set of remaining instruction nodes.

For a page size of 8, the following pagination is obtained:

- \( P_{14} \): I1, C25, I, K1, C50, C2, J, C15
- \( P_{15} \): I2, I3, C1, C10, M(1), M(2)
- \( P_{16} - P_{21} \): M(3) - M(50)
- \( P_{22} - P_{84} \): N(1) - N(500).

For comparing performances, we only use the total number of page faults generated as the criterion, and assume a fixed partition size. (For a fixed partition size the space-time cost becomes proportional to the number of page faults alone.) The data pagination assumed for B & C's algorithm is to order data sequentially in the order of their first reference and then paginate sequentially. Thus the data pagination for their algorithm is:

- \( P_{15} \): K1, I1, I2, I3, C15, C1, C50 I
- \( P_{16} \): C2, C25, J, C10, M(1), M(2)
- \( P_{17} - P_{22} \): M(3) - M(50)
- \( P_{23} - P_{85} \): N(1) - N(500)

The instruction pagination for the two algorithms was given in Figure 2.2.5.
Figure 2.3.2 - Node Trace

B&C: \[ P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15} P_{16} P_{17} P_{18} P_{19} P_{20} P_{21} P_{22} P_{23} P_{24} \]

\[ T_1 \]

\[ T_2 \]

\[ T_3 \]

Figure 2.3.3 - Page Traces
### Table 2.3.3

<table>
<thead>
<tr>
<th>Resident Set Size</th>
<th>Page Faults B &amp; C</th>
<th>Page Faults OUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23,758</td>
<td>19,958</td>
</tr>
<tr>
<td>2</td>
<td>23,156</td>
<td>19,439</td>
</tr>
<tr>
<td>3</td>
<td>21,832</td>
<td>12,507</td>
</tr>
<tr>
<td>4</td>
<td>13,074</td>
<td>1,816</td>
</tr>
<tr>
<td>5</td>
<td>3,044</td>
<td>387</td>
</tr>
<tr>
<td>6</td>
<td>140</td>
<td>138</td>
</tr>
</tbody>
</table>

**Figure 2.3.4 - Page Fault vs Res. Set Size**
Consider the node trace (Figure 2.3.2) for the program execution (for the values of variables assumed previously). The corresponding page traces for two paginations are given in Figure 2.3.3. The page faults produced for the various resident set sizes, and the corresponding graphs are shown in Table 2.3.3 and Figure 2.3.4, respectively.

Our pagination schemes perform much better for this example. The main reason (besides ones mentioned for instruction pagination) is that for the loop which is executed most often, our pagination produces fewer data pages—by keeping all data required by the corresponding locality together, whence fewer distinct pages are referenced in those loops. If, as for the B & C, the data were paginated sequentially (or alphabetically), then the number of distinct pages required by each loop would be larger thereby increasing the total number of page faults for a given resident set size.
2.4 COLLECTION OF A PRIORI INFORMATION

In this section we consider the determination of a priori information (i.e. "localities") for programs so that this information can be made available to the operating system (memory management algorithms) during the execution of programs. The information we need include the following:

- determination of localities
- mean and variance in locality size
- locality membership.

2.4.1 Determination of Localities

Earlier (Section 1.1.4) while defining localities based upon SCR, we stated two constraints each locality must satisfy; i.e.,

I. a locality size must be less than the maximum partition size allowed for the program.

II. the expected execution time of a locality must exceed the quantum of time allotted to each program.\(^1\)

If \(S_{\text{max}}\) is the maximum partition size allowed for the program, then SCR which are admissible as localities as per condition I must satisfy the following condition on SCR size \(S_{\text{SCR}}\), i.e.,

\[
S_{\text{SCR}} = I_{\text{SCR}} + D_{\text{SCR}} < S_{\text{max}} \tag{2.4.1}
\]

\(^1\)For batch multiprogramming systems the quantum time may be replaced by average execution interval between page faults required for efficient operation of the system.
where

\[ I_{\text{SCR}} = \text{the number of instruction pages in the SCR.} \]
\[ D_{\text{SCR}} = \text{the number of data pages connected to the instruction} \]
\[ \text{pages constituting the SCR. Here for data bases and} \]
\[ \text{matrices of size greater than one page, we only include} \]
\[ \text{one page per such structure. (We assume that for each} \]
\[ \text{such structure, normally a single page would be} \]
\[ \text{required at a time.)} \]

Condition I (2.4.1) puts an upper bound on the size of localities, and condition II provides a lower bound on the execution time (and indirectly on size). We would like to choose the localities large enough so that the execution time spent in each locality is adequate to justify the overheads (determination and maintenance of locality information). Thus SCRs whose estimated execution time satisfies condition II, and whose size requirement does not violate condition I may be designated as localities.

We base the determination of localities on the page graph mainly, and use SCRs in the program graph to provide approximate estimates of execution times for pages of the page graph. For this we first determine approximate execution time for each SCR in program graph. We shall assume that instruction nodes belonging to an SCR are equally important, and hence will be executed with equal frequency. (This is reasonable only because of simplicity, otherwise for a better estimate we have to use some "analysis of algorithms" procedure—e.g. Section 2.3.4.1 of Knuth [43].) We also assume that an SCR nested within the
SCR under consideration is represented by a single node, and every time the outer SCR is executed, the inner SCR nodes are executed k times, where k is the expected execution frequency of the inner SCR each time it is entered.

For SCR\(_i\), we define:

\[
\tau_{SCR_i}(1) = \begin{cases} 
\text{execution time spent in SCR}_i \text{ if the nodes} \\
\text{of SCR}_i \text{ are executed exactly once}
\end{cases}
\]

\[
\tilde{\tau}_{SCR_i} = \begin{cases} 
\text{expected execution time spent in SCR}_i \\
\text{each time SCR}_i \text{ is entered}
\end{cases}
\]

\[
= \tau_{SCR_i}(1) \times \begin{cases} 
\text{average execution frequency of} \\
\text{SCR}_i \text{ each time it is entered}
\end{cases}
\]

(2.4.2)

Since we consider all nodes of an SCR equally important, we may write:

\[
\tau_{SCR_i}(1) = \begin{cases} 
\text{number of instruction} \\
\text{nodes in SCR}_i \text{ each} \\
\text{instruction node}
\end{cases}
\times \begin{cases} 
\text{average execution time for} \\
\text{each instruction node}
\end{cases}
\]

If SCR\(_i\) contains another SCR\(_j\), then

\[
\tau_{SCR_i}(1) = \left[ \begin{cases} 
\text{number of instruction} \\
\text{nodes in SCR}_i \text{ not} \\
\text{common to SCR}_j
\end{cases} \right] \times \begin{cases} 
\text{average execution} \\
\text{time per} \\
\text{instruction node}
\end{cases} + \tilde{\tau}_{SCR_j}
\]

The determination of estimated execution times of SCRs thus reduces to the problem of estimation of execution frequency of each SCR, every time the SCR is entered. For this purpose we classify SCRs into three types and estimate their execution frequencies accordingly.

(a) The SCRs for which the execution frequency is known.

Examples for this type are "DO loops" with known value of number of iterations each time the loop is executed.

(b) The SCRs for which the execution frequency can be
estimated. Examples for this type are "DO loops" where the number of iterations depend on a variable whose value can be approximated, or the SCRs which refer to a matrix (or an array) the dimensions of which can be used as the estimate. Other heuristics, or programmer "advice," may provide reasonable estimates.

(c) The SCRs for which no information is available about the execution frequencies. For such SCRs, we cannot use the timing information for deciding localities. Then we have to depend only upon the size information and use condition I to determine localities. This is discussed further below.

If we know the approximate execution times for all SCRs, we can determine a rough estimate of execution time for each page in the instruction graph as follows.

\[ t_{\text{page}_i} = \left\{ \begin{array}{ll}
\text{estimated execution time for } i^{\text{th}} \text{ page,} \\
\text{each time page } i \text{ is entered} 
\end{array} \right. \]

\[ r_i = \sum_{j=1}^{r_i} \text{SCR}_j \cdot \text{nodes on } i^{\text{th}} \text{ page not common to any } \text{SCR}_j \cdot \text{average node execution time} \]

where \( r_i \) is the number of disjoint SCRs within the \( i^{\text{th}} \) page.

The rough estimate of execution time for SCRs in the page graph (of size greater than a page) may be obtained by considering the SCRs in the page graph which also correspond to SCRs on the node graph. Note that the page graph SCRs which do not correspond to program graph SCRs are not significant because pages corresponding to those SCRs will not be executed iteratively and hence do not correspond to a locality. Such SCRs do not represent execution for significant amounts of time. Let
PSCR\textsubscript{i} denote the \textit{i}th significant SCR in the page graph; an SCR on a page graph is a PSCR only if it corresponds to an SCR on the node graph.

Then

\[ \tilde{t}_{\text{PSCR}_i} = \text{approximate execution time for } \text{i}^{th} \text{ PSCR in a page graph} \]

\[ n_i = \sum_{j=1}^{n_i} \left\{ \text{approximate execution} \right\} \times \left\{ \text{frequency of } \text{PSCR}_i \right\} \]

where \( n_i \) is the number of pages contained in PSCR\textsubscript{i}.

The localities among pages of the page graph may be assigned using procedure 'a' given below. The procedure uses a "nestedness" tree for the PSCRs. In the nestedness tree a PSCR\textsubscript{i} which is immediately nested within PSCR\textsubscript{j} appears as a (direct) descendent of the node corresponding to PSCR\textsubscript{j}. The procedure to assign localities is then as follows:

a.1 Apply the following steps to all nodes (PSCRs) at the same level (distance from the "root"), starting with the highest level, until all nodes of the tree have been considered.

a.1.1 If the node (PSCR) does not contain a descendent node which has been assigned to a locality, then:

a.1.1.1 If the PSCR corresponding to that node satisfies conditions I and II, assign a locality to the PSCR.

a.1.1.2 Otherwise, if two or more descendents of the same node in the tree satisfy conditions I and II jointly, assign a locality to the smallest such set of PSCRs.

a.1.2 Otherwise, consider the next node in the tree.

This algorithm determines the smallest PSCRs satisfying condition II.
To reduce the number of localities, the algorithm may be modified to combine localities smaller than $S_{\text{max}}$ whenever possible. It should also be noted that not all PSCRs need be assigned to localities (i.e. those too large in size or too brief in execution time).

An alternative to the above procedure is to establish a minimum locality size, $S_{\text{min}}$; e.g. for efficient operation of pre-loading systems a minimum size of pre-load set is recommended (see Section 4.3.2). Then the localities may be determined from the page graph as PSCRs whose size lies between $S_{\text{max}}$ and $S_{\text{min}}$. For nested PSCRs, the largest PSCR which does not violate condition I may be designated a locality. The choice, here, may also depend upon the program type, if known: e.g. for a compiler, larger PSCRs may be chosen as localities since approximate execution time for each page is small; for looping programs smaller PSCRs may qualify for locality assignment.

2.4.2 Mean and Variance in Locality Size

A mean and variance is associated with each locality for use in the Bayesian allocation algorithms of Chapter 4. In Bayesian algorithms the memory to be allocated to a program is re-estimated at various intervals, known as execution (or reallocation) intervals (chosen equal to the 'quantum' time, for example). Thus the mean locality size estimated should reflect as correctly as possible the program's memory needs during the interval that a locality is in control.

If the $j^{\text{th}}$ locality consists of $r_j$ disjoint PSCRs (in the instruction page graph), then its mean locality size is:
If the jth locality consists of two or more nested PSCRs, then we proceed as follows. We recall that a locality for nested PSCRs was assigned to the PSCR whose estimated execution time exceeds the quantum, while that of all its immediately nested PSCRs is less than quantum time. An example is where PSCR1 is nested within PSCR2, and

\[ \tilde{r}_{PSCR1} < q \quad \text{and} \quad \tilde{r}_{PSCR2} > q. \]

To find the mean locality size two cases arise:

**Case I.** \( t_{PSCR2}(1) \leq q \). Then all the pages corresponding to PSCR2 would be needed in every quantum, and hence \( m_j' = s_{PSCR2} \).

**Case II.** \( t_{PSCR2}(1) > q \). In this case we have to estimate the mean locality size. We consider a weighted sum, weighted according to the execution time estimates, to find the mean. For our example, we can estimate \( m_j' \) as follows:

\[
m_j' = \frac{\frac{t_{PSCR1}}{q} + \frac{(s_{PSCR2} - s_{PSCR1})(t_{PSCR2} - t_{PSCR1})}{q}}{t_{PSCR2}(1)/q}
\]

The variance is similarly estimated as:

\[
\sigma_j'^2 = \frac{(m_j' - s_{PSCR1})^2 \tilde{r}_{PSCR1} + (m_j' - s_{PSCR2} + s_{PSCR1})^2 \left( t_{PSCR2}(1) - \tilde{r}_{PSCR1} \right)}{t_{PSCR2}(1)/q}
\]

The means and variances for other localities with nested PSCRs may be similarly obtained.
For localities with no execution time information the mean and variance may be simply estimated by finding the average size requirements for the PSCRs nested within the locality. Thus,

\[ m^i_j = \frac{1}{r_j} \sum_{i=1}^{r_j} S_{PSCR_i} \]

\[ \sigma^2_j = \frac{1}{r_j} \sum_{i=1}^{r_j} (m^i_j - S_{PSCR_i})^2 \]

where \( j^{th} \) locality contains \( r_j \) nested PSCRs.

### 2.4.3 Locality Membership

Once the localities have been designated, the membership of each locality may be determined and recorded for use by memory management algorithms (to be discussed in later chapters). We recommend keeping a list of "essential" pages for each locality. The essential pages are the pages (of a locality) which have a higher probability of being referenced once the process enters the locality. We consider only the essential pages because the locality itself may be very large, especially if it contains a large matrix or data structure. Most of the pages of such structures may not be referenced every time locality is executed, or if so then only once or twice. The pages which have higher probability of reference are the instruction pages and data pages (corresponding to simple variables alone) which are referenced by the nodes of the SCRs constituting the locality. (Note that there may be some pages in \( D_{PSCR_j} \), the data locality corresponding to PSCR\( j \), which are not referenced by any node of the largest SCR in the locality.) We do not recommend keeping any data pages corresponding to large matri-
ces or data structures on the list because it is not easy to determine which of its pages will be needed next. The list is maintained along with the page map information to facilitate its use.

To signal change in localities, we recommend maintaining information about "entry" pages along with the page map information. An entry page is an instruction page in a locality which may be referenced by instructions outside the locality. The entry page should be designated such that each entry page belongs to only one locality, while a locality may have more than one such page. Each entry page has to be marked to indicate the locality (number) to which the page belongs. An alternative could be to keep similar information for "exit" pages of each locality. The main problem with such a scheme is that when a locality is exited, the program may not immediately enter another locality. We would still need to know when a new locality is being entered.

2.4.4 Discussion

In the earlier discussion for determination of localities we must point out that many times the estimate of execution time obtained may be quite different from the actual value; especially in case III of determination of execution frequencies. The true execution time of such a locality may be much smaller or much larger than the quantum time. If the actual execution time is much smaller, the major effect of this would be in the extra runtime overhead required due to an excessive number of locality changes. If on the other hand the actual execution time is much larger than the quantum time, then the allocation algorithm proposed (in Chapter 4) will degenerate into historical
algorithms since the savings due to recognition of new localities will be minimal. It is better to err in the latter direction (i.e. to make "conservative" estimates of localities) as the cost would be primarily in wasted compile time calculations rather than run time degradation of performance.

We illustrate the above by considering the example of B & C, examined earlier in this chapter (Figure 1.1.1). The SCRs of "importance" and their respective membership in program graph nodes, as well as their execution times (in terms of the number of instructions executed) are shown in Table 2.4.1.

Table 2.4.1 - Est. Exec. Time

<table>
<thead>
<tr>
<th>SCR No.</th>
<th>Membership (nodes)</th>
<th>t_{SCR}(1)</th>
<th>$\bar{t}_{SCR}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>{5-12}</td>
<td>44</td>
<td>220</td>
</tr>
<tr>
<td>30</td>
<td>{14-17}</td>
<td>27</td>
<td>1350</td>
</tr>
<tr>
<td>32</td>
<td>{5-18}</td>
<td>1579</td>
<td>18948</td>
</tr>
<tr>
<td>22</td>
<td>{22}</td>
<td>8</td>
<td>280</td>
</tr>
<tr>
<td>29</td>
<td>{21-23}</td>
<td>288</td>
<td>2880</td>
</tr>
</tbody>
</table>

Note that we only consider the instruction nodes here since we are mainly interested in the timing information. (The values of $\bar{t}_{SCR}$ shown in this table are the actual values for the page trace considered earlier.)

Assume a quantum time $q = 10,000$ (instruction executions). If we choose $SCR_{31}$ and $SCR_{30}$ as localities (by erroneously estimating their
execution time, or if no estimates of time were available) then, when
the program is in SCR\textsubscript{32}, SCR\textsubscript{31} and SCR\textsubscript{30} will be called repeatedly
within the same quantum causing a large number of \((\approx \left[ \frac{10,000}{1579} \right] \times 2)\)
locality changes. If SCR\textsubscript{32} were chosen as a locality instead, the
number of locality changes produced will be reasonably low and memory
requirements during execution of SCR\textsubscript{32} can be predicted with consider­
able accuracy.

If we assume a quantum \(q = 100\), and choose SCR\textsubscript{29} as a locality,
then, this locality will remain in the memory for a large number of
quantums \((\approx \left[ \frac{2880}{100} \right])\) and the size of SCR\textsubscript{29} does not correspond to the
memory requirements for a quantum during its execution. SCR\textsubscript{22} provides
a better estimate for memory requirements, for a quantum, during the
execution of SCR\textsubscript{29}.

The number of such degenerate cases, in which such gross inaccurate
estimates occur, should be small. In general there will be a large
proportion of programs for which even the rough estimates would suffice
as appropriate indicators of program memory demands. An example of
this is the page trace used for simulation in Section 4.4. The page
trace is for execution of a compiler, and only the information about
the page SCRs of the page graph is used for assigning localities. The
simulation results show that the gains produced by use of such a priori
information are substantial. In any case, the excessive number of
locality changes, which is the main cause for deterioration of predic­
tive algorithms, can be averted by disabling the locality change
recognition mechanism for a fraction of a quantum immediately after a
new locality is loaded (cf. Section 4.4).
We should also point out another problem that may arise if all SCRs of the program graph exceed the $S_{max}$ bound. The memory allocation algorithms (of Chapter 4), in such a case, will not produce any improvements in the performance over existing algorithms, because memory requirements would never be satisfied. No memory allocation algorithm would perform satisfactorily in this "thrashing" situation; to avoid this, $S_{max}$ must be sufficiently large.
CHAPTER 3

Replacement

The replacement problem is that of choosing a page to replace upon a page fault. If the page selected to be replaced is one which will be referenced furthest away in time, then the optimal decision has been made (cf. Belady [22]). However, this decision cannot be made realizably since we cannot determine exactly what the future program behavior is going to be. In that case we regard an optimal replacement decision as one which, based upon the information available, makes the best decision possible according to a given criterion. The information available may be of various kinds, e.g. historical, probabilistic, structural. In this chapter we consider development of optimal replacement algorithms, first based on probabilistic information. Since, as we argued earlier, probabilistic information is unreliable, we next develop an optimal replacement scheme based upon the knowledge of structural information. Finally we develop a class of suboptimal replacement algorithms which make use of structural as well as historical information, because page structure information alone may not be sufficient to arrive at a unique decision and history of reference information is easily obtainable and widely used in the design of replacement algorithms.
3.1 OPTIMAL REPLACEMENT USING PROBABILISTIC INFORMATION

3.1.1 An Optimal Replacement Algorithm

We develop here an optimal replacement algorithm for Markovian assumptions on the program model. The optimality is in terms of minimization of expected cost, or the expected number of page faults incurred by the program operating under demand paging. Demand paging is modeled as a stochastic control process, as in [51]. Demand paging, then, is governed by the following state-transition equation:

\[ x_{i+1} = g(x_i, y_i, r_i); \quad i \geq 1 \quad (3.1.1) \]

where

- \( x_i = \) set of pages in memory at stage \( i \) (the resident set),
- \( y_i = \) the page chosen for replacement, if \( r_i \notin x_i \),
- \( r_i = \) the page referenced at stage \( i \).

The function 'g' is defined as follows:

\[ g(x, P_1, P_2) = \begin{cases} x, & \text{if } P_2 \in x \\ x \cup \{P_2\} - \{P_1\}, & \text{if } P_2 \notin x \text{ and } P_1 \in x \quad (3.1.2) \\ x \cup \{P_2\}, & \text{if } P_2 \notin x \text{ and } P_1 \notin x \end{cases} \]

i.e. demanded page \( P_2 \) replaces the page \( P_1 \) in the resident set \( x \).

Our objective is to select the optimal replacement decision sequence \( \{y_1^*(r_1), y_2^*(r_2), \ldots\} \) where \( y_i^*(r_i) \) is the optimal page to be replaced at stage \( i \), if page \( r_i \notin x_i \) is referenced. In the text to follow, \( y_i(r_i) \) will be abbreviated to \( y_i \) except where explicit representation is warranted. Also, we shall use the simplified notation \( x + P_2 \)
to denote \( x \cup \{P_2\} \) and \( x - P_2 \) to denote \( x - \{P_2\} \). The optimal sequence is chosen to minimize the **cost functional**

\[
H(\{x_1, x_2, \ldots\}, \{y_1, y_2, \ldots\}, \{r_1, r_2, \ldots\})
\]

for a page reference string \( \{r_1, r_2, \ldots\} \) and initial condition \( r_0 \in x_1 \).

We assume that \( H \) is separable so that

\[
H = \sum_i h(i) \gamma(x_i, y_i, r_i);
\]

where

\[
h(i) = \text{cost of a page exception at the } i^{th} \text{ stage}
\]

\[
\gamma(x_i, y_i, r_i) = \begin{cases} 
1, & \text{if } r_i \notin x_i \\
0, & \text{otherwise}
\end{cases}
\]

This assumption asserts that the cost associated with going from state \( x_i \) to \( x_{i+1} \) depends only upon \( x_i \), \( y_i \), and \( r_i \) (an independence assumption). For paging systems a normal assumption made is

\[
h(i) = 1, \forall i.
\]

The cost \( H \) then represents the total number of page faults encountered in processing the reference string \( \{r_1, r_2, \ldots\} \) under the state transition map defined by (3.1.1) and (3.1.2).

The page replacement problem may then be defined as that of minimizing the expected cost function, i.e.

\[
\text{Exp}_{\{r_1\}} \left[ \min_{\{y_i\}} \left\{ \sum_i h(i) \gamma(x_i, y_i, r_i) \right\} \right]
\]

The problem so formulated can be solved by means of dynamic programming.
Define
\[ f_N'(x_k, k) = \begin{cases} \text{minimum expected cost associated with an N-stage} \\ \text{(reference string assumed of length N) replacement} \\ \text{decision process, in state } x_k \text{ at stage } k, \text{ averaged} \\ \text{over all possible pages referenced at kth step.} \end{cases} \]

Then it can be established [51] that the dynamic programming functional equation is given as:
\[
\begin{align*}
\mathbb{E} \left[ \min_{r_k} \left\{ h(k)\gamma(x_k, y_k, r_k) + f_N'(g(x_k, y_k, r_k), k + 1) \right\} \right] \\
= f_N'(x_k, k)
\end{align*}
\]
(3.1.5)

with terminal condition \( f_N'(x, N+1) = 0 \). Functional equation (3.1.5) yields the minimum expected cost at any stage \( k \) of the replacement process.

In [51], (3.1.4) was solved under the assumption of independent page references. The model assumed here for generation of reference string \( \{r_k\} \) is that of a stationary Markov process, commonly used in the literature [28,52]. The reference string is assumed to be obtained from a Markov process with stationary transition probabilities \( [p_{ij}] \) given as:
\[
p_{ij} = \begin{cases} \text{probability of referencing page } j \text{ by } \text{last page referenced is page } i \end{cases}, \quad 1 \leq i, j \leq n
\]
\[ n = \text{total no. of states in the Markov process.} \]

Then we can write (3.1.5) as:
\[
\begin{align*}
f_N'(x_k, k) = \sum_{r_k} \sum_{r_{k-1}} \sum_{r_k} \left[ \min_{y_k} \left\{ h(k)\gamma(x_k, y_k, r_k) + f_N'(g(x_k, y_k, r_k), k + 1) \right\} \right] p_{r_k-1}\]
\end{align*}
\]

The optimal replacement decision \( y_k^* \) for a resident set \( x_k \) and referenced page \( r_k (r_k \neq x_k) \), is given by
\[
y_k^* = \text{ARG} \left[ \min_{y_k} \left\{ h(k)\gamma(x_k, y_k, r_k) + f_N'(g(x_k, y_k, r_k), k + 1) \right\} \right]
\]
where the notation \( \text{ARG} \left[ \min_x \{ F(x) \} \right] \) designates the value of \( x \) that minimizes
F(x). (The dependence of \( y^*_k \) on \( x_k \) and \( r_k \) is not shown explicitly for notational simplicity.)

We define another cost function \( f_N(x_k, r_k, k) \), defined for each referenced page \( r_k \), as follows:

\[
f_N(x_k, r_k, k) = \begin{cases} 
\text{minimum expected cost with an N-stage replacement decision process, at stage } \\
\text{if referenced page is } r_k \\
h(k) + \min \left\{ \sum_{y_k} p_{r_k} r_{k+1} f_N(x_{k+1}, r_{k+1}, k+1) \right\}, & \text{if } r_k \neq x_k \\
\sum_{r_k} p_{r_k} r_{k+1} f_N(x_k, r_{k+1}, k+1), & \text{otherwise}
\end{cases}
\] (3.1.6)

with the end condition

\[
f_N(x, r_N, N) = \begin{cases} 
h(N), & \text{if } r_N \neq x \\
0, & \text{otherwise}
\end{cases}
\] (3.1.7)

The two cost functions are related as follows:

\[
f_N^i(x_k, k) = \sum_{r_k} p_{r_k} r_{k-1} f_N(x_k, r_k, k)
\]

We will now derive an expression for cost \( f_N(x_1, r_1, 1) \) for \( N=3 \), and use that to obtain the optimal replacement decision at the current step (i.e. \( k=1 \)), assuming \( r_1 \neq x_1 \). For \( N=3 \) and \( r_1 \neq x_1 \) we can rewrite expression (3.1.6) as

\[
f_N(x_1, r_1, 1) = h(1) + \min \left\{ \sum_{y_1} p_{r_1} r_{2} f_3(x_2, r_2, 2) \right\}
\]

where \( x_2 = g(x_1, y_1, r_1) = x_1 - y_1 + r_1 \).

Applying the functional equation (3.1.6) recurrently, we obtain
. \ f_3(x_1, r_1, 1) = h(1) + \min_{y_1} \left( \sum_{r_2 \in x_2} \sum_{r_3 \notin x_2} p_{r_1 r_2} p_{r_2 r_3} f_3(x_2, r_3, 3) + \right.
\left. \sum_{r_2 \notin x_2} p_{r_1 r_2} (h(2) + \min_{y_2} \sum_{r_3 \notin x_2} p_{r_2 r_3} f_3(x_3, r_3, 3)) \right) \ (3.1.8)

where \ x_3 = g(x_2, y_2, r_2) = x_2 - y_2 + r_2.

Using the end condition (3.1.7), we have

\ \ f_3(x_1, r_1, 1) = h(1) + \min_{y_1} \left( \sum_{r_2 \in x_2} \sum_{r_3 \notin x_2} p_{r_1 r_2} p_{r_2 r_3} h(3) + \right.
\left. \sum_{r_2 \notin x_2} p_{r_1 r_2} (h(2) + \min_{y_2} \sum_{r_3 \notin x_2} p_{r_2 r_3} h(3)) \right)

Then the optimal page \ y_1^* \ to be replaced, if \ r_1 \ (r_1 \notin x_1) \ is referenced, is given by

\ \ y_1^* = \text{ARG} \left\{ \min_{y_1} \left( \sum_{r_2 \in x_2} \sum_{r_3 \notin x_2} p_{r_1 r_2} p_{r_2 r_3} h(3) + \sum_{r_2 \notin x_2} p_{r_1 r_2} h(2) + \right. \right.
\left. \left. \sum_{r_2 \notin x_2} p_{r_1 r_2} \left( \min_{y_2} \sum_{r_3 \notin x_2} p_{r_2 r_3} h(3) \right) \right) \right\}

Using relation \ x_3 = x_2 - y_2 + r_2 \ to expand the last term, we obtain

\ \ y_1^* = \text{ARG} \left\{ \min_{y_1} \left( \sum_{r_2 \in x_2} \sum_{r_3 \notin x_2} p_{r_1 r_2} p_{r_2 r_3} h(3) + \sum_{r_2 \notin x_2} p_{r_1 r_2} h(2) + \right. \right.
\left. \left. \sum_{r_2 \notin x_2} p_{r_1 r_2} \left( \min_{y_2} \sum_{r_3 \notin x_2} p_{r_2 r_3} + p_{r_2 y_2} - p_{r_2 r_2} \right) h(3) \right) \right\}

Combining similar terms (in first and third terms), we have

\ \ y_1^* = \text{ARG} \left\{ \min_{y_1} \left( \sum_{r_2 \in x_2} \sum_{r_3 \notin x_2} p_{r_1 r_2} p_{r_2 r_3} h(3) + \sum_{r_2 \notin x_2} p_{r_1 r_2} h(2) + \right. \right.
\left. \left. \sum_{r_2 \notin x_2} p_{r_1 r_2} \left( \min_{y_2} \left\{ p_{r_2 y_2} - p_{r_2 r_2} \right\} \right) h(3) \right) \right\}
Using relation $x_2 = x_1 - y_1 + r_1$ to expand the first, second, and third terms, we obtain

$$y_1^* = \text{ARG} \left\{ \min \left\{ \sum_{y_1} p_{r_1 y_1} (\sum_{y_2} p_{r_2 y_2} \sum_{y_3} p_{r_3 y_3} \cdot h(3)) + \sum_{y_1} p_{r_1 y_1} h(2) \right\} \right\} (3.1.9)$$

where $y_2^*$ is the optimal page to be replaced at the next page fault, if

$$r_2$$

is the next referenced page

$$= \text{ARG} \left\{ \min_{y_2 \in x_2} \{ p_{r_2 y_2} \} \right\}$$

Considering terms dependent on choice of $y_1$ in (3.1.9), we obtain

$$y_1^* = \text{ARG} \left\{ \min \left\{ \sum_{y_1} p_{r_1 y_1} \sum_{y_2} p_{r_2 y_2} \sum_{y_3} p_{r_3 y_3} \cdot h(3)) + p_{r_1 y_1} h(2) \right\} \right\} (3.1.10)$$

where $x_2' = x_1 + r_1 - y_2^*(y_1)$, and

$$y_2^*(y_1)$$

is the optimal page to be replaced if $y_1$ is referenced.

If we assume that the cost of a page exception is the same at any stage, i.e. $h(k) = 1, \forall k$, then

$$y_1^* = \text{ARG} \left\{ \min \left\{ \sum_{y_1} p_{r_1 y_1} \sum_{y_2} p_{r_2 y_2} \sum_{y_3} p_{r_3 y_3} \cdot h(3)) + p_{r_1 y_1} h(2) \right\} \right\} (3.1.11)$$

The optimal replacement decision at any step thus minimizes the total probability of referencing the page to be replaced along all paths of length $< N (=3)$, plus the probability of referencing $y_2^*$ through a nonresident page, plus the probability of referencing nonresident pages following an immediate re-reference of the page to be replaced. The
latter term may be neglected if the probability that any nonresident page is referenced by a resident page, even by the one to be replaced, is small. This is true if programs satisfy the locality property (i.e., localities change membership slowly) and memory is managed accordingly. More specifically, we assume
\[ \sum_{z \notin x} p_{yz} \ll 1 \] (3.1.12)
where \( x \) is any resident set and \( y \in x \). This is almost certainly true since \( p_{yy} \) (the probability that a page references itself) is generally \( \gg \frac{1}{z} \). We note that (3.1.12) is equivalent to the assumption that page fault probabilities are uniformly small (the uniformity being a consequence of the stationarity of our Markov model), or for example that (say) ten non-faulting page references can be expected in between consecutive page faults. (If instead a "run" of page faults should occur, sub-optimal individual replacement decisions can be tolerated so long as the set of such replacements is optimal.) Relation (3.1.12) also implies that \( p_{r_1 r_2} \), for \( r_2 \notin x + r_1 \), is small, so the third term in (3.1.11) may also be neglected. Utilizing (3.1.12), expression (3.1.11) reduces to
\[ y_1^* = \text{ARG} \left\{ \min_{y_1} \left( p_{r_1 y_1} + \sum_{r_2 \notin y_1} p_{r_1 r_2} p_{r_2 y_1} \right) \right\} \] (3.1.13)

Generalizing for arbitrary \( N \) (see Appendix C), we obtain
\[ y_1^* = \text{ARG} \left\{ \min_{y_1} \left( p_{r_1 y_1} + \sum_{r_2 \notin y_1} p_{r_1 r_2} p_{r_2 y_1} + \sum_{r_3 \notin y_1} p_{r_3 y_1} \right) \right\} \] (3.1.14)
or,

\[ y^*_1 = \begin{cases} \text{page } y_1 \text{ which has the largest probability of} \\
\text{reference (LPR) from } r_1 \text{ to } y_1 \text{ along all paths} \\
\text{of length } < N \text{ (not including self loops).} \end{cases} \]

Thus we propose the following class of replacement algorithms:

"The LPR(N) replacement policy replaces a page with the least 
probability of reference, along all paths of length less than 
a parameter \( N \), from the currently referenced page."

LPR differs from LTP \([22]\), which chooses \( y_1 \) to minimize \( p_{r_1 y_1} \), in that 
paths of length greater than one are considered; in fact, \( \text{LTP} = \text{LPR}(2) \). 
LPR replacement is optimal under the Markovian and locality assumptions. 
An upper bound on \( N \) is the number of pages of a program, since longer 
paths contain a loop and only the first reference to a page is signifi-
cant.

### 3.1.2 Determination of Reference Probabilities

The parameter \( N \) in the above may be discarded by adopting an 
absorbing Markov chain model; i.e. we assume there is at least one 
absorbing state reachable from all other states. The Markov process 
will then have finite absorption "time" \( T \), and in effect \( N \) can be 
chosen greater than \( T \).

The page-graph transition probability matrix \( P \) may be partitioned 
as shown:

\[
P = \begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} = \begin{pmatrix} P_{11} & P_{12} \\ 0 & I \end{pmatrix}
\]

where \( P_{11} \) is an \( n_1 \times n_1 \) matrix corresponding to the \( n_1 \) transient states
of the Markov process, and I is an identity matrix corresponding to
absorbing states. Then the reference frequency matrix \( F \) (for transient
states alone) is given by

\[
F = [I - P_{ll}]^{-1} \tag{3.1.15}
\]

by virtue of the absorbing chain assumption [64], and

\[
\hat{f}_{ij} = \text{probability of referencing page } j \text{ from page } i, \text{ along }
\text{all paths of length } \geq 0, \text{ before absorption}
\]

\[
= \frac{f_{ij}}{\sum_j f_{ij}}
\]

These probabilities (rather than \( p_{ij} \)) may be used for general LPR re-
placement; i.e. LPR policy would replace a page \( j \) in the resident set
with minimal value of \( \hat{f}_{ij} \), where \( i \) is the currently referenced page.

Note that the above method gives the reference probabilities for
transient pages only. Let us consider what happens when an absorbing
page is referenced. Let \( x_1 \) be the resident set, and \( r_1 \) be the page
referred (an absorbing page). Since \( r_1 \) is an absorbing page, it is
immaterial what page is replaced, because the program execution is
terminated at the next step, and whatever page is replaced does not
change the cost. Also, an absorbing page cannot be a member of the
resident set at any time (except at termination), since we have assumed
that page loading is on demand only.
3.2 OPTIMAL REPLACEMENT USING STRUCTURAL INFORMATION

As we have observed earlier, algorithms based on Markovian assumptions are unrealistic because of their dependence upon probabilities of transition. Hence we propose basing optimal replacement decision on some information which is more dependable and easily obtainable (and also is not data dependent). The structure of a program graph provides such information since the structure does not change, such information is reliable. (At times programs may behave as if its structure were dynamic, so our static assumption is a conservative one.) We shall develop an optimal replacement algorithm which uses the program connectivity information alone. Also, we shall not assume that the cost incurred at the current step (as in (3.1.10)), h(1), and cost incurred k steps into future, h(1+k), k ≥ 1, are the same. This is because page faults that may arise in distant future, relative to a replacement decision now, should be "discounted." We thus introduce a page fault cost function which is monotonically decreasing as a function of the future distance; we define

\[ h(1) = 1 \]
\[ h(k) = h(k-1) / \alpha_{k-1} = \prod_{i=1}^{k-1} (1/\alpha_i), \quad k > 1 \quad (3.2.1) \]

The only constraint we place on the values of \( \alpha_k \) are as follows;

\[ \alpha_k \geq \alpha > 1, \forall k \quad (3.2.2) \]

We next develop a class of replacement algorithms using these assumptions.
3.2.1 INR - Interval of Next Reference Replacement Algorithm

The optimality criterion developed in Section 3.1 depends upon the knowledge of transition probabilities. Since we are assuming here use only of structural information about the program, we must adopt an equally likely probability of transition model, i.e., we assume

\[ \hat{P}_{ij} = \text{probability of transition from page } i \text{ to page } j \]

\[ = \frac{c_{ij}}{\sum_j c_{ij}} \quad (3.2.3) \]

An approximate value for this probability is

\[ \hat{P}_{ij} = \frac{c_{ij}}{\bar{n}} \quad (3.2.4) \]

where \( \bar{n} \) is the average number of pages (immediate successors) connected to a page. Substituting \( \hat{P}_{ij} \)'s for \( P_{ij} \)'s in expression (3.1.10) we obtain

\[ y^* = \text{ARG} \left[ \min \left\{ \sum_{y_1} \hat{P}_{r_1 y_1} \hat{P}_{r_2 y_1} h(3) + \hat{P}_{r_1 y_1} h(2) + \sum_{r_2 x_1} \hat{P}_{r_1 r_2 x_1 \bar{y}_2} h(3) + \hat{P}_{r_1 y_1} \sum_{r_3 x_2} \hat{P}_{r_1 r_3 x_2 \bar{y}_3} h(3) \right\} \right] (3.2.5) \]

The assumption of the locality property (made earlier in Section 3.1) implies that the number of nonresident pages connected to a resident page, as compared to the number of remaining pages connected to the same resident page, is small. In particular, we assume

\[ \sum_{z \notin x} \hat{P}_{yz} << \sum_{z \in x} \hat{P}_{yz} < 1 \quad (3.2.6) \]

where \( x \) is any resident set and \( y \notin x \). If the number of nonresident pages connected to \( y \) is large, the locality property may be violated, since then the likelihood of referencing a nonresident page would be large. Hence using (3.2.6) (and since \( h(2) \geq h(3) \)), the last term in (3.2.5) may be neglected in comparison to the second term. Relation (3.2.6) also
implies that the third term may be neglected in comparison to the first term. (Note that many of \( p_{r_2y_2} \) in the third term will be zero if the average resident set size is greater than the average number of pages connected to any page, since \( y_2^{*} \) is chosen such as to minimize \( p_{r_2y_2} \) among resident pages.) Thus utilizing (3.2.6), expression (3.2.5) reduces to

\[
y_1^{*} = \text{ARG} \left( \min \left\{ y_1^{(p_{r_1y_1}h(2)} + \sum_{r_2} p_{r_1r_2} p_{r_2y_2} h(3) \right\} \right)
\]

For an arbitrary \( N \), we obtain (similar to (3.1.12)),

\[
y_1^{*} = \text{ARG} \left( \min \left\{ y_1^{(p_{r_1y_1}h(2)} + \sum_{r_2} p_{r_1r_2} p_{r_2y_2} h(3) + \ldots \right. \right.
\]

\[
\ldots \left. + \sum_{r_{N-1}} p_{r_{N-1}y_{N-1}} h(N) \right\} \right)
\]

(3.2.7)

If we define \( \beta_i(y_1) \) as

\[
\beta_i(y_1) = \text{sum of transition probabilities along all paths of length } i \text{ from } r_1 \text{ to } y_1
\]

\[
= \sum_{r_2} p_{r_1r_2} \sum_{r_3} p_{r_2r_3} \ldots \sum_{r_{i-1}} p_{r_{i-1}y_{i-1}} p_{r_{i-1}y_1}
\]

then expression (3.2.7) reduces to

\[
y_1^{*} = \text{ARG} \left( \min_{y_1} \left\{ \beta_i(y_1)h(i+1) \right\} \right)
\]

(3.2.8)

Thus the optimal page is one which minimizes this weighted sum of transition probabilities along all paths of length \( < N \).

We will obtain an approximate bound on each term, \( \beta_i(y_1) \), and use those bounded values to derive an expression for the optimal page.

Using assumption (3.2.3) and approximation (3.2.4), we can rewrite \( \beta_i(y_1) \) as
\[
\beta_1(y_1) = \frac{1}{r_1} \cdot \sum_{r_1} c_{r_1} r_2 \cdot \frac{1}{r_2} \cdot \sum_{r_2} c_{r_2} r_3 \cdot \frac{1}{r_3} \cdot \sum_{r_3} c_{r_3} r_1
\]

where

\[
\delta(r_1, y_1, i) = \begin{cases} 
1, & \text{if } \exists \text{ a path of length } i \text{ from } r_1 \text{ to } y_1 \\
0, & \text{otherwise.}
\end{cases}
\]

Using the approximation (3.2.4) we can write

\[
\sum_{r_j} \frac{\gamma}{r_j} \cdot \sum_{r_j} c_{r_j} r_{j-1} \cdot \frac{1}{r_{j-1}} \cdot \sum_{r_{j-1}} c_{r_{j-1}} r_j = (1 - \frac{1}{n}) < 1,
\]

because \(\gamma > 0\), assuming each page has a self loop since generally at least two or more instructions will be executed from a page before another page is referenced. Hence the value of \(\beta_1(y_1)\) may be approximated as;

\[
\beta_1(y_1) \approx (1 - \frac{1}{n})^{i-1} \frac{\delta(r_1, y_1, i)}{n} \tag{3.2.10}
\]

The value of \(\beta_1(y_1)\) estimated above is an upper bound and its actual value will be smaller. This is because as \(i\) becomes large, the number of nodes of set \(\{r_i\}\), in expression (3.2.9), that have an edge incident with \(y_1\) will be smaller than cardinality of set \(\{r_i\}\) (see Figure 3.2.1).

Hence the actual contribution of the last factor will be smaller.

Now, for \(\alpha \geq 2\) we know that the inequality

\[
\frac{1}{\alpha^{i-1}} > \sum_{j=1}^{N-1} \frac{1}{\alpha^j}
\]

is true for any \(i\) and \(N\). Therefore for \(n > 1\) and for some value of \(\alpha\), \(2 > \alpha > 1\), the following will also be true;

\[
(1 - \frac{1}{n})^{i-2} \frac{1}{\alpha^{i-1}} > \sum_{j=1}^{N-1} (1 - \frac{1}{n})^{j-1} \frac{1}{\alpha^j}
\]
Figure 3.2.1 - An Example
Also, since $\alpha_\ell \geq \alpha$ for all $\ell$ by assumption (3.2.2), it follows that

$$(1 - \frac{1}{N})^{i-2} \prod_{\ell=1}^{i-1} \frac{1}{\alpha_\ell} > \sum_{j=1}^{N-1} \frac{1}{N} (1 - \frac{1}{N})^{j-1} \frac{1}{\sum_{\ell=1}^{j} \alpha_\ell}$$

or

$$(1 - \frac{1}{N})^{i-2} h(i) > \sum_{j=1}^{N-1} \beta_j (y_1) h(j+1) \quad (3.2.11)$$

If $\delta(r_1, y_1, i) = 1$, then we can write (using (3.2.10))

$$\beta_{i-1} (y_1) h(i) > \sum_{j=1}^{N-1} \beta_j (y_1) h(j+1) \quad (3.2.12)$$

(The value of $\alpha$ which needs to be assumed for (3.2.12) to hold will be smaller than that required for (3.2.11) to hold, due to introduction of $\beta_\ell$'s and also because some of the $\beta_j$'s on the right hand side may be zero as all paths of length $j \geq i+1$ may not exist between $r_1$ and $y_1$.)

Using (3.2.12), the optimal replacement decision can now be stated as follows:

$$y_1^* = \text{the page which minimizes the sum in expression (3.2.8)}$$

$$= \text{page which has largest } \ell \exists \delta(r_1, y_1, j) = 0 \text{ for each } j < \ell.$$  \hfill (3.2.13)

i.e., the optimal page is one which has the maximum value of the interval of first next reference from page $r_1$. We call the replacement algorithm which replaces a page according to (3.2.13) the Interval of Next Reference (INR) replacement algorithm. The INR replacement policy is the following:

"On a page fault (referencing $r_1$), replace a page $y_1^*$ in the resident set $x_1$, such that the interval (path length) of (first) next reference to $y_1^*$ from $r_1$ is maximal."
For determining the optimal page to replace, we need consider simple paths alone (i.e. paths with no loops), since for every path of length \( l \) with a loop, there exists a path of length \(< l\) from node \( r_1 \) to \( y_1 \), and we are interested only in the first next reference. The maximum length up to which we need to investigate the paths depends upon \( N \). A prudent choice for \( N \) is the length of the longest simple path between any two nodes of the program graph, because all paths of greater length would necessarily contain loops. A more reasonable value of \( N \), if we assume the "locality" model for program execution, is the maximum length of a simple path in any locality, because while the program is in a particular locality, paths to other localities are of lesser importance. We give a listing of a program, in Appendix I, which takes as its input the resident set and the currently referenced page and outputs a page with maximum INR suitable for replacement.

For any choice of \( N \), the INR replacement policy may not discriminate among several candidate pages—i.e. ties could result. Since connectivity information alone may yield a number of pages which are candidates for replacement, to choose among these pages, we have to make use of other criteria. As discussed previously, the use of historical information is most appropriate here, because future program behavior depends upon past history as well. Thus we introduce, next, a class of replacement algorithms which uses both structural and historical information for making a replacement decision. We do not claim any optimality for these but in general they are expected to provide improved performance over existing practical replacement algorithms, and yet permit efficient implementation.
3.3 EXEMPTIVE REPLACEMENT ALGORITHMS

In this section we discuss a class of replacement algorithms which are sub-optimal but more practical. For the INR replacement algorithms, very large values of $N$ may have to be considered to resolve ambiguities in replacement decisions, and even then not necessarily completely. We note, however, that the terms $\beta_i(y_i)$ decrease very rapidly with increasing path lengths (cf. (3.2.10)) and thus their effect on the sum becomes negligible. In such a case, considering larger values of $N$ for resolving conflicts among pages will not make much difference except to introduce more computations. Thus it is reasonable to set a threshold value for $N$ up to which a search for paths is made. This reduces the computations required for locating paths of various lengths for different pages. However, if the value of $N$ is restricted, the structural information alone is even more likely to offer a multitude of pages which are suitable for replacement. Thus we have to use some other criteria to decide which among these is the best page to replace. We employ the history of reference information to aid in making this decision. Historical information is an important predictor of program behavior and has been used very widely in the design of replacement algorithms.

For a given $N$, every page in memory which has a path of minimum length $\geq N$, from the currently referenced page, $r_1$, is a candidate for replacement. The pages in memory which are reachable from $r_1$ within paths of length $\leq N$ are probable pages to be used in near future and hence should not be replaced, i.e., should be granted exemptions (hence the name "exemptive" algorithms). Thus if a set of pages $y_{i,1} \leq x_1$, all

---

109
have paths of length < N from $r_1$, then the pages in $y_1'$ are not considered for replacement, so long as $y_1'' \neq \emptyset$, where $y_1'' = x_1 - y_1'$. Of the remaining pages, i.e., $y_1''$, the page chosen for replacement is one which is historically least likely to be referenced in near future. (If $y_1'' = \emptyset$, then historical information alone is used to make the replacement decision.) Thus a mixture of structural and historical information is used for replacement. The extent of structural information used and the kind of historical information employed for ranking pages determines one class of exemptive replacement algorithms.

For $N = 2$, only the simple one-step connectivity information is used to grant exemptions. Only pages which are not directly connected to the currently referenced page are considered for replacement. Thus a catastrophe of a replaced page being immediately referenced is avoided. Such algorithms are fairly practical also because the main information used is historical information and the simple connectivity information required for granting exemptions can be easily obtained from the compilation stage and kept in the page tables. For this case, i.e., $N = 2$, we can have two types of exemptive algorithms

- Safe Exemptive Replacement Algorithms
- Non-safe Exemptive Replacement Algorithms

**Safe Exemptive Algorithms.** For these algorithms the pages are ranked by an underlying historical replacement algorithm and only the lowest two pages are considered for replacement, depending upon which one of the two is directly\(^1\) connected to $r_1$. If none or both are

\(^1\)This is different from one proposed by Lew [31], which exempts a page even if it is indirectly connected to $r_1$. 

connected, then the lowest ranked page is replaced. In the following section we show that, for Markovian assumptions, a Safe Exemptive Algorithm is as good or better than the corresponding historical algorithm.

**Non-Safe Exemptive Algorithms.** For these algorithms all the pages not connected to the page $r_1$, are considered for replacement. The ranking used corresponds to some underlying historical replacement algorithm. If all pages in $x_1$, the resident set, are connected to $r_1$, then the page to be replaced is chosen using the historical information alone. This scheme does not necessarily produce a lower cost than corresponding non-exemptive algorithms, but in general shows considerable improvements, as shown by simulation results (given in Section 3.2.2).

A variation of the exemption policy is one where the pages of the current "locality" are granted exemptions against replacement. Thus only pages which belong to the current locality will remain in the main memory. This replacement policy is very useful for pre-loading memory allocation algorithms (cf. Section 4.3.2) where the essential pages of the locality are pre-loaded when the program enters the corresponding locality. These pages should be exempted from replacement for some time, because they have a great likelihood of being referenced and if they are replaced, more page traffic will result.

3.3.1 **Evaluation of Safe Exemptive Algorithms**

We evaluate the performance of Safe Exemptive algorithms using the formulation and criterion developed by Franklin and Gupta (FG[52]). An absorbing Markov chain model is used and an expected number of page
faults criterion is adopted. The model developed by FG is equivalent to one by Ingargiola and Korsh (IK[28]). In particular transition probability matrix $Z$ of FG is the same as $Q$ matrix of IK, with the absorbing substates in the latter deleted. The $Z$ matrix corresponds to transitions among substates defined by FG as triples $(s,q,k)$, where $s$ is the memory state (the set of resident pages), $q$ is the control state (ranking of pages, depending upon the replacement algorithm), and $k$ is the page currently in control. To evaluate the Safe Exemptive algorithms, the $Z$ matrix for the underlying non-exemptive algorithm has to be modified according to the replacement decision taking connectivity into consideration. The substates for the exemptive algorithm and the non-exemptive algorithm will be identical, since both use the same historical algorithm to rank pages. The expected page fault rates (EPFR) are given by

$$
\text{EPFR}_{\text{N.E.}} = \text{expected page fault rate for the non-exemptive algorithm} \\
= \sum_{i=1}^{N} p_{\text{N.E.}}(ss_i) \pi(ss_i)
$$

$$
\text{EPFR}_{\text{EXEM}} = \text{expected page fault rate for the exemptive algorithm} \\
= \sum_{i=1}^{N} p_{\text{EXEM}}(ss_i) \pi(ss_i)
$$

where

$N = \text{number of states},$

$\pi(ss_i) = \text{probability of page faulting from substate } ss_i,$

$p(ss_i) = \text{probability of being in substate } ss_i \text{ (assuming } ss_1 \text{ is the initial state)}$

$$
= f_{1j} / \sum_j f_{1j},
$$

$f_{ij} = \{ \text{expected no. of times } ss_j \text{ is entered | initial substate } ss(0) = ss_i \}$
$$F = [f_{ij}] = [I - Z]^{-1},$$ the frequency of enterence matrix.

(The inverse exists since $Z$ is associated with an absorbing Markov chain.)

Note that $\pi(ss_i)$ values are independent of the replacement rule since the probability of faulting from any state depends only upon the resident set and the control page. The difference between the replacement algorithms lies in which state is entered upon a page fault, which depends upon page rankings; thus any changes produced in EPFR are due to changes in the $Z$ matrix. Since a change of a single replacement decision between exemptive and non-exemptive rules would only interchange two elements of a row of $Z$ matrix, we shall explore the effect of one such interchange on the resulting $F$ matrix.

Let us assume that the elements $z_{ij}$ and $z_{ij_1}$ are interchanged due to a change in replacement decision. Note that one of these elements is a zero. Let $z_{ij_1} = 0$ for non-exemptive, and $z_{ij} = 0$ for exemptive. Let $Z^*$ be the matrix obtained from $Z$ by interchanging of elements $z_{ij}$ and $z_{ij_1}$ of $Z$ and $F^*$ denote its corresponding frequency matrix.

To facilitate comparison of the expected page fault rates for the exemptive and non-exemptive schemes, we shall obtain $F$ and $F^*$ in terms of a common matrix $B$. First define $Y = [I - Z]$ and $Y^* = [I - Z^*]$, both non-singular. Then $Y^*$ is obtained from $Y$ by interchange of elements $y_{ij}$ and $y_{ij_1}$.

Let $\mathbf{g}^i = i$th column of $N \times N$ identity matrix, and

$\mathbf{k}^j = j$th row of an $N \times N$ identity matrix.

Then if $y_{ij} = -z_{ij} = -a$, we can write

$$Y = A + \mathbf{g}^i(-a)\mathbf{k}^j$$
\[ Y^* = A + g^1 (-a) g^1 \]  

(3.3.1)

where A is the same as Y (or \( Y^* \)) with \((i,j)\)th and \((i,j_1)\)th elements set equal to zero. Using a result from matrix algebra [63], namely

\[ P^{-1} = Q^{-1} - Q^{-1} G (K^{-1} G + H^{-1})^{-1} K Q^{-1}, \]

if \( P - Q = GHK \) and \( P, Q \) are non-singular, we obtain from (3.3.1)

\[ Y^{-1} = A^{-1} - A^{-1} g^1 [x^1 A^{-1} g^1 + 1/(-a)]^{-1} x^1 A^{-1} \]

\[ = A^{-1} - A^{-1} [A_{j_1}^{-1} - 1/a]^{-1} A_{j_1}. \]

where 

\[ A_{i_1}^{-1} = \text{ith column of } A^{-1}, \]

\[ A_{j_1}^{-1} = \text{jth row of } A^{-1}. \]

From the definitions of \( F \) and \( Y \), we obtain

\[ F = Y^{-1} = A^{-1} + (1/a - A_{j_1}^{-1})^{-1} [A_{i_1}^{-1} A_{j_1}^{-1}] \]

\[ F^* = (Y^*)^{-1} = A^{-1} + (1/a - A_{j_1}^{-1})^{-1} [A_{i_1}^{-1} A_{j_1}^{-1}] \]

Replace matrix \( A^{-1} \) by matrix \( B \) (for convenience)

\[ F = B + (1/a - b_{j_1})^{-1} [B_{i_1} B_{j_1}] \]

\[ F^* = B + (1/a - b_{j_1})^{-1} [B_{i_1} B_{j_1}] \]

\( F \) and \( F^* \), the frequency of entrance matrices for the non-exemptive and exemptive schemes, respectively, have now been obtained in terms of a (common) matrix \( B \). The expected page fault rates for the two algorithms can be estimated as follows. Assuming \( s_{s_1} \) is the initial substate,

\[ \text{EPFR}_{\text{N.E.}} = F_{1_1} \prod_j f_{1j} \]

\[ = \{ B_{1_1} + b_{1_1} (1/a - b_{j_1})^{-1} B_{j_1} \} \prod_j f_{1j} \]  

(3.3.3)

\[ \text{EPFR}_{\text{EXEM}} = F^*_{1_1} \prod_j f_{1j} \]

\[ = \{ B_{1_1} + b_{1_1} (1/a - b_{j_1})^{-1} B_{j_1} \} \prod_j f_{1j} \]
where $\mathbf{I}$ = column vector of substate page fault transition probs.

The frequency matrix $B$ is similar to the matrix $F$ (or $F^*$) and it is obtained in a similar manner as $F$ (or $F^*$) except with element $(i,j)$ of the $Z$ matrix (or $(i,j_1)$ of the $Z^*$ matrix) set equal to zero. This corresponds to eliminating the transition from substate $s_{Si}$ to $s_{Sj}$ (or $s_{Sj_1}$). In the substate transition diagram (corresponding to matrix $B$) the transitions out of the substate $j$ and $j_1$ are to the same substates and with the same probabilities, therefore the sum of elements of row $j$ will be identical to the sum of elements of row $j_1$. The two rows will have the same values in corresponding columns except in columns $j$ and $j_1$. Thus

$$b_{jj_1} + b_{jj} = b_{j_1j_1} + b_{j_1j}$$

(3.3.4)

This is true for all replacement algorithms in which the last ranked page does not attain a higher rank without being referenced. This holds for all stack replacement algorithms as well as for FIFO and others which satisfy the above statement. Thus we can write

$$\text{EPFR}_{\text{N.E.}} = \{C + (b_{1j} + b_{1i}(1/a - b_{ji})^{-1}b_{jj}) \pi_j + (b_{1j_1} + b_{1i}(1/a - b_{ji})^{-1}b_{jj_1}) \pi_{j_1}\} \sum_j f_{lj}$$

$$\text{EPFR}_{\text{EXEM}} = \{C + (b_{1j} + b_{1i}(1/a - b_{ji})^{-1}b_{jj}) \pi_j + (b_{1j_1} + b_{1i}(1/a - b_{ji})^{-1}b_{jj_1}) \pi_{j_1}\} \sum_j f_{lj}$$

where $C = \sum_{i \neq j, j_1} f_{li} \pi_i = \sum_{i \neq j, j_1} f_{li^*} \pi_i$

Since $b_{ji} = b_{j_1i}$, let $b_{1i}(1/a - b_{ji})^{-1} = b_{1i}(1/a - b_{j_1i})^{-1} = k > 0$. Then for $\text{EPFR}_{\text{N.E.}} > \text{EPFR}_{\text{EXEM}}$ to be true, we require that
\[ b_{jj} \pi_j + b_{jj_1} \pi_{j_1} > b_{j_1 j} \pi_j + b_{j_1 j_1} \pi_{j_1} \]  \hfill (3.3.5)

That the probability of page fault in substate \( S_{S_j} \) is smaller than in substate \( S_{j_1} \) (i.e., \( \pi_{j_1} > \pi_j \)) is shown below. The two substates differ in memory membership by a single page alone.

Let \( y_j \) be the page replaced in transition from \( S_{S_j} \) to \( S_{S_j} \),
\[ y_{j_1} \] be the page replaced in transition from \( S_{S_j} \) to \( S_{S_j} \), and
\( x \) be the referenced page which caused this transition.

Since \( y_j \) is directly connected to \( x \) (hence not replaced by the exemptive algorithm) a reference to \( y_j \) will not cause a page fault. The probability of referencing \( y_j \) is zero, therefore
\[ \pi_j = \pi_{j_1} + p_{xy_j} \]
where \( p_{xy_j} \) = probability of referencing \( y_j \) when \( x \) is in control, and
\[ p_{xy_j} > 0 \] by definition of exemptive algorithms. Hence \( \pi_j > \pi_{j_1} \).

The inequality (3.3.5) may therefore be rewritten as
\[ (b_{jj} + b_{jj_1}) \pi_j + b_{jj} p_{xy_j} > (b_{j_1 j} + b_{j_1 j_1}) \pi_{j_1} + b_{j_1 j} p_{xy_j} \]  \hfill (3.3.6)

Simplifying (using (3.3.4)) we obtain,
\[ \text{EPFR}_{N.E.} > \text{EPFR}_{EXEM} \text{ if } b_{jj} > b_{j_1 j} \]  \hfill (3.3.7)

If substate \( S_{j_1} \) is not reachable from \( S_{S_j} \), \( b_{j_1 j} \) will be zero and condition (3.3.7) will be satisfied; \( b_{jj} > 0 \) since. Otherwise, we proceed as follows:

The substate transition graph may be reduced to the graph shown in Figure 3.3.1 where
\[ p_{\ell m} = \text{probability of transition from } S_{S_{\ell}} \text{ to } S_{S_m} \text{ through paths of length 1 or more, such that } \ell \neq m \text{ and } \ell, m \in \{j, j_1\}. \]
$S_k = \text{set of all absorbing states}$

Reduced Substate Trans. Diagram

*Figure 3.3.1*
\[ p_{\ell \ell} = \begin{cases} 
\text{probability of transition from } s_{\ell} \text{ to } s_{\ell}', \text{ through paths of length 1 or more not including } s_{m}, \text{ such that } \ell \neq m, \\
\ell, m \in \{j, j_1\}. 
\end{cases} \]

\[ p_{\ell k} = \text{probability of transition to an absorbing state from } s_{\ell}
\]

without going through \( s_{m} \), such that \( \ell \neq m, \ell, m \in \{j, j_1\} \)

Then, since original transition graph is assumed to be Markovian, we may write

\[ p_{jj} + p_{jj_1} + p_{jk} = p_{j_1j} + p_{j_1j_1} + p_{j_1k} = 1. \quad (3.3.8) \]

Using Markovian analysis [53], and Mason's rule for finding transmissions [62], we get

\[ b_{jj} = (1 - \tilde{p}_{jj_1}) / \Delta \quad (3.3.9) \]
\[ b_{j_1j} = \tilde{p}_{j_1j} / \Delta \quad (3.3.10) \]

where \( \Delta = 1 - (\tilde{p}_{jj} + \tilde{p}_{jj_1} + \tilde{p}_{j_1j} \tilde{p}_{j_1j_1} + \tilde{p}_{jj} \tilde{p}_{jj_1}) \).

Using (3.3.8) we obtain

\[ \Delta = \tilde{p}_{j_1j} \tilde{p}_{jk} + \tilde{p}_{j_1k} (\tilde{p}_{jj_1} + \tilde{p}_{jk}) \]

Expressions (3.3.9) and (3.3.10) hold only if \( \Delta > 0 \), which is true if \( \tilde{p}_{j_1j} \) and \( \tilde{p}_{jk} > 0 \), or if \( \tilde{p}_{j_1k} \) and \( \tilde{p}_{jj_1} > 0 \), or if \( \tilde{p}_{j_1k} \) and \( \tilde{p}_{jk} > 0 \)

\[ (3.3.11) \]

Condition (3.3.11) may be stated as, \( \Delta > 0 \) if there is a path, with probability > 0, of transition from \( s_{j_1} \) or \( s_{j} \) to an absorbing state.

This is true since both \( s_{j} \) and \( s_{j_1} \) are non-absorbing by assumption.

That is,

\[ \tilde{p}_{jk} > 0 \text{ and } \tilde{p}_{j_1k} > 0 \quad (3.3.12) \]

So \( \Delta > 0 \).

Using (3.3.8), (3.3.9), (3.3.10), and (3.3.12) we conclude
Hence condition (3.3.7) is established, i.e., \( EPFR_{N.E.} > EPFR_{EXEM} \).

Thus safe exemptive replacement algorithm will produce a lesser number of expected page faults (for Markovian assumptions) than the underlying non-exemptive replacement algorithm.

### 3.3.2 Simulation

To test the effectiveness of non-safe exemptive algorithms, we rely on simulation because theoretical analysis is intractable without additional assumptions.

Our simulation could have been carried out in a number of ways:

a) simulate the entire system, including the page trace generation process.

b) simulate only the memory management parts and use page traces generated by actual runs of the program.

c) simulate via actual implementation of algorithms on a virtual memory system.

The third possibility was eliminated because of the cost of such an experiment. We used a mixture of (a) and (b) for simulation, because the amount of actual page trace data available was limited.

For (a) the programs were simulated employing a technique reported by Thorington and Irwin [23], with some modifications. Thorington and Irwin generate a program trace by simulating programs constructed by generating a sequence of instructions of one of twelve types. The instruction types range from those that are simply executed (do not require any data, and do not alter the normal instruction sequence), to
those that involve loops, and those with conditional and unconditional jumps to random or constant address. A program is then constructed by assigning probabilities to each instruction type and using a pseudo-random number generator to obtain a sequence of successive instruction types. The program is then paginated sequentially. The page containing the instruction and the data referenced by it then gives a partial page trace (corresponding to execution of that instruction). The modifications introduced by us are:

1) introduction of a new type of instruction to simulate references to elements of a matrix;

2) introduction of a transition probability matrix among instruction types instead of absolute probabilities used by them for each instruction type. This was done to make constructed programs more realistic because in practice normally only a few instruction types may follow any instruction type;

3) for looping instructions, the address to which loop back occurs is such that the corresponding instruction is known (i.e. already generated). This is done to simulate "DO loop" type constructs more realistically.

Typical results of simulations are summarized in Table 3.3.1. The programs considered were as follows:

A. the simulation program itself

B. a predominantly random-looping program

C. a predominantly sequential program

D. predominantly random data references
### Table 3.3.1

Page Fault Comparison of Replacement Algorithms

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>PAGE SIZE</th>
<th>PROGRAM MEMORY SIZE</th>
<th>LRU</th>
<th>ELRU</th>
<th>LFU</th>
<th>ELFU</th>
<th>FIFO</th>
<th>EFIFO</th>
<th>SIM</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>512</td>
<td>100</td>
<td>10</td>
<td>926</td>
<td>914</td>
<td>1022</td>
<td>849</td>
<td>1005</td>
<td>980</td>
<td>931</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>756</td>
<td>723</td>
<td>831</td>
<td>671</td>
<td>768</td>
<td>725</td>
<td>756</td>
<td>558</td>
<td>606</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>601</td>
<td>581</td>
<td>645</td>
<td>524</td>
<td>624</td>
<td>567</td>
<td>606</td>
<td>454</td>
<td>606</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>478</td>
<td>442</td>
<td>485</td>
<td>390</td>
<td>489</td>
<td>417</td>
<td>459</td>
<td>369</td>
<td>459</td>
</tr>
<tr>
<td>B</td>
<td>128</td>
<td>60</td>
<td>5</td>
<td>1141</td>
<td>1140</td>
<td>1401</td>
<td>1206</td>
<td>1386</td>
<td>1341</td>
<td>1141</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>957</td>
<td>952</td>
<td>1088</td>
<td>1016</td>
<td>1038</td>
<td>1022</td>
<td>963</td>
<td>619</td>
<td>619</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>657</td>
<td>647</td>
<td>724</td>
<td>617</td>
<td>695</td>
<td>602</td>
<td>657</td>
<td>393</td>
<td>393</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>359</td>
<td>306</td>
<td>397</td>
<td>291</td>
<td>407</td>
<td>347</td>
<td>657</td>
<td>393</td>
<td>393</td>
</tr>
<tr>
<td>C</td>
<td>128</td>
<td>90</td>
<td>10</td>
<td>1004</td>
<td>1003</td>
<td>1103</td>
<td>1035</td>
<td>1089</td>
<td>1061</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>725</td>
<td>704</td>
<td>780</td>
<td>685</td>
<td>724</td>
<td>682</td>
<td>689</td>
<td>390</td>
<td>390</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>395</td>
<td>347</td>
<td>458</td>
<td>347</td>
<td>407</td>
<td>289</td>
<td>374</td>
<td>229</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>116</td>
<td>100</td>
<td>158</td>
<td>138</td>
<td>166</td>
<td>105</td>
<td>93</td>
<td>87</td>
<td>87</td>
</tr>
<tr>
<td>D</td>
<td>512</td>
<td>100</td>
<td>10</td>
<td>320</td>
<td>319</td>
<td>309</td>
<td>290</td>
<td>326</td>
<td>323</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>250</td>
<td>246</td>
<td>265</td>
<td>240</td>
<td>270</td>
<td>256</td>
<td>240</td>
<td>224</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>193</td>
<td>190</td>
<td>201</td>
<td>181</td>
<td>204</td>
<td>183</td>
<td>206</td>
<td>190</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>173</td>
<td>159</td>
<td>161</td>
<td>152</td>
<td>166</td>
<td>163</td>
<td>172</td>
<td>169</td>
<td>169</td>
</tr>
<tr>
<td>E</td>
<td>512</td>
<td>75</td>
<td>10</td>
<td>264</td>
<td>263</td>
<td>226</td>
<td>221</td>
<td>259</td>
<td>266</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>206</td>
<td>206</td>
<td>190</td>
<td>177</td>
<td>240</td>
<td>233</td>
<td>207</td>
<td>181</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>185</td>
<td>184</td>
<td>158</td>
<td>155</td>
<td>179</td>
<td>178</td>
<td>158</td>
<td>141</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>173</td>
<td>173</td>
<td>137</td>
<td>132</td>
<td>158</td>
<td>156</td>
<td>137</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>F</td>
<td>1024</td>
<td>232</td>
<td>6</td>
<td>806</td>
<td>800</td>
<td>992</td>
<td>856</td>
<td>800</td>
<td>772</td>
<td>806</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>730</td>
<td>723</td>
<td>845</td>
<td>722</td>
<td>749</td>
<td>725</td>
<td>730</td>
<td>558</td>
<td>558</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>604</td>
<td>603</td>
<td>682</td>
<td>619</td>
<td>668</td>
<td>640</td>
<td>609</td>
<td>497</td>
<td>497</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>551</td>
<td>543</td>
<td>584</td>
<td>553</td>
<td>556</td>
<td>538</td>
<td>551</td>
<td>497</td>
<td>497</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>473</td>
<td>473</td>
<td>477</td>
<td>463</td>
<td>453</td>
<td>434</td>
<td>407</td>
<td>481</td>
<td>481</td>
</tr>
</tbody>
</table>
E. an actual matrix-multiplication program
F. an actual compiler page trace.

The replacement algorithms studied were the following:
1. LRU - least recently used
2. ELRU - exemptive LRU
3. LFU - least frequently used
4. ELFU - exemptive LFU
5. FIFO - first in first out
6. EFIFO - exemptive FIFO
7. SIM - the adaptive replacement algorithm in [23]
8. OPT - modified Belady algorithm [22] with limited lookahead.

The number of references processed were 2500 for programs A to E and 1500 for program F.

The simulations were performed for non-safe exemption policies. The results show that degradation is generally rare and mild on occurrence. The performance of exemptive algorithms relative to non-exemptive algorithms shows considerable improvements for ELFU and EFIFO algorithms. In most cases the performance of ELRU improves as the amount of main memory allocated to the program increases. However the improvements are not generally very large. This may be expected in case of the simulated programs, because each instruction page is connected to a large number of pages. This reduces the effectiveness of exemptive algorithms because:

* if the number of connected pages is so large that all pages in memory are connected to the instruction page at the time of a replacement decision, the page chosen for replacement
is the same as that for the non-exemptive algorithm.

- a large number of pages have to be provided exemptions even though the probability of referencing most of the connected pages is small.

A solution to above problem is to limit the number of pages connected to each instruction page. This may be obtained as a by-product of reordering and pagination (see Chapter 2), because the effort there is to put code and data connected to each instruction page onto a small number of pages. If connectivity is high even after reordering, one of the alternatives considered in the following section may be used.

3.3.3 Implementation

Implementation of exemptive replacement algorithms requires that additional space for connectivity matrix be allocated in the page containing page mapping information. These matrices are boolean, so 'bit-arrays' may be used. However, this space saving technique would result in increased access times for connectivity information. Alternatively, if matrices are sparse, a list storage technique can be used because each page generally only directly references a small subset of the set of pages for the program. If a page is connected to a large number of pages of the program, we may consider connectivity information to certain "essential" pages alone. These essential pages may be:

- those pages which belong to the same locality as the page whose connectivity is being examined
- those pages which have higher probability of reference (cf. Lew [54])
those pages which have other paths (of length two or more) from the page under consideration (besides being directly connected).

Thus if the number of pages connected to a page is limited, it is not unreasonable to associate with each page a list of such pages. This information may be stored with the rest of the page mapping information for each instruction page. At a page fault, this connectivity list is examined, along with the historical information to determine a page for replacement. The connectivity list examined is that corresponding to the demanded page and is used to grant exemption to those pages. The time required for these checks will be insignificant relative to the fetch time (especially if there is a separate microprocessor for memory management), and particularly when compared with the cost of a wrong decision. This overhead time can be recouped if page faults are indeed reduced. Thus implementation of the exemption policy is reasonably practical.
CHAPTER 4
Memory Allocation

We discuss here solutions to problems in memory allocation as outlined in Section 1.4. First we consider the determination of optimal partition sizes based upon the knowledge of program requirements. The problem of optimal partitioning is formulated very suitably for application of the Lagrange method of optimization. We consider application of these ideas to determine the partitions to be allocated among programs in the main memory.

Next we consider the estimation of locality sizes based upon Bayesian decision theory. We make use of a loss function to arrive at optimal estimates of locality sizes. We also consider the question of membership of localities. This is considered in greater detail with respect to systems where localities are loaded in as a batch before processes gain control, i.e. pre-loading systems. We make use of Bayesian ideas and the locality model of program behavior to obtain better estimates of the pre-load sets. Finally we consider problems of implementation of these pre-loading schemes in relation to the BCC 500 virtual memory computer system. We show through simulation that implementation would lead to large savings in loading costs.
4.1 OPTIMAL STORAGE PARTITIONING

We discuss here the problem of partitioning storage (main memory) among processes in an "optimal" manner. Optimality must be relative to a specific quantitative criterion, to be defined. We assume that an estimate of the "cost" of limiting the number of pages allocated to a program is available. Specifically, we assume that a criterion function, \( c_k(q) \), a measure of the performance of program \( k \) in a partition size of \( q \), can be derived. Further we assume that the cost \( c_k(q) \), for each program \( k \), is independent of \( c_j(q) \), for any other program \( j \neq k \). This is in contrast to other approaches where programs are considered identical (although independent). For these assumptions, the partitioning problem may be formally stated as:

\[
\min_{\{q_k\}} \sum_{k=1}^{N} c_k(q_k), \quad \text{subject to} \quad \sum_{k=1}^{N} q_k = M \quad (4.1.1)
\]

(and possibly other constraints representing the multiprogramming or scheduling environment), where \( q_k \) is the partition size of program \( k \), \( N \) is the total number of competing programs, and \( M \) is the total number of available pages in memory. A formal solution to the optimization problem requires knowledge of the cost function \( c_k(q) \) for varying allotment sizes \( q \). The various cost functions that may be used include:

(A) The probability that the locality size of program \( k \) exceeds its partition size \( q \), i.e. \( \Pr[x > q] \), where \( x \) is a random variable representing the locality size. Performance measures used by other authors (e.g. Coffman and Ryan [32]) are variations of this cost function. This cost function requires modeling program behavior as a
random process, using statistics on the locality size (the process type, mean and variance) for cost function formulation.

(B) The page fault rate of the program $k$ running in a partition size $q$. This is a common and convenient criterion, but its estimation may be difficult. One means of obtaining the expected page fault rate is by monitoring program performance over a range of sizes. An alternative with considerably less overhead involves measuring the page fault rates for only certain allotments and obtaining the entire cost curve through curve fitting [65]: Empirical studies [40] have shown that the page-fault versus allotment curve may be approximated by a sequence of line segments of increasing negative slopes, with prominent breakpoints corresponding to the strongly connected subsets of pages; thus if the page fault rates for these breakpoints can be estimated, the entire cost curve can be obtained. For stack replacement algorithms, a variation may be to use the stack distance frequencies as the cost. These may be determined through monitoring, as for example in Chu and Opderbeck [24].

Note that formulation in (4.1.1) also allows programs to have zero allocation in the main memory. For such cases the cost function $c_k(0)$ must be defined. To determine $c_k(0)$ the cost of letting a process sit in secondary memory without executing must be estimated. This cost will generally vary with time, and depends on the priority of a process also;
e.g. if a process has been waiting too long (in secondary memory) its priority goes up and eventually it replaces a process in main memory with a lower priority. Thus the cost for a zero allocation is time-varying and its value depends upon scheduling considerations (besides the program parameters). For example, if process \( k \) has a priority \( p_k(t) \) at time \( t \), then the cost for zero allocation \( c_k(0,t) \), may be defined as

\[
c_k(0,t) = \hat{c}_k(0) \times p_k(t)/p_{\text{min}}
\]

where

\[
\hat{c}_k(0) = \begin{align*}
\text{the cost of zero allocation assuming all processes have equal priority}, \\
p_{\text{min}} = \text{minimum priority any process may have}
\end{align*}
\]

The effect of this cost function will be that processes with higher priority will be given memory allocations at the cost of low priority processes. The cost \( \hat{c}_k(0) \) may be chosen by extrapolating the cost curves obtained for non-zero allocations, or \( \hat{c}_k(0) \) may be set equal to \( \hat{c}_k(a_k) \) where \( a_k \) is the minimum partition size that should be allocated to process \( k \) for execution with some efficiency (without an excessive page fault activity). The latter choice will force the low priority process to execute only when a minimum partition size can be allocated. The higher priority processes may have to be allocated partition sizes smaller than the minimum to satisfy response requirements for such processes. The number of such high priority processes which occupy main memory together may be limited due to other considerations, e.g. process table size, minimum number of pages to be allocated to a process.
(2 or more), etc. For general cost functions, the optimal storage partitioning problem has been solved by mathematical programming techniques [30].

Here we discuss a method to obtain a closed form optimal solution (of (4.1.1)) assuming continuous and differentiable cost functions, employing the Lagrange method of optimization. Consider the performance measure, \( P = f(x_1, x_2, \ldots, x_n) \). \( P \) is to be minimized with respect to a selection of \( x_i \)'s, where the \( x_i \)'s are restricted to satisfy an equality constraint, \( c_1 = g(x_1, x_2, \ldots, x_n) \). The assumptions required for the Lagrange method to be applicable are:

i) functions \( f \) and \( g \) are continuous and have continuous derivatives.

ii) the constraint equation admits to a set of real solutions.

iii) \( \frac{\partial^2 f}{\partial x_i^2} \bigg|_{x_i=x_i^*} > 0 \), where \( x_i^* \) is the minimum value of \( x_i \), \( \forall i \).

The solution then proceeds as follows. First an augmented function \( f_a \) is formed, \( f_a = f + \lambda(g-c_1) \), where \( \lambda \) is the Lagrange multiplier and is independent of the \( x_i \)'s. Next it is observed that the stationary points of \( f_a \) are required to satisfy,

\[
\frac{\partial f_a}{\partial x_i} = 0, \text{ for } i = 1, 2, 3, \ldots, n. \tag{4.1.2}
\]

Equation (4.1.2) and the constraint give \((n+1)\) equations in \((n+1)\) variables \((x_i \)'s and \( \lambda \)), which if independent give a unique solution of \( x_i \)'s that minimizes the performance \( P \).

For the optimal allocation problem, we have to minimize \( P \) with respect to the \( q_k(t) \)'s, where \( q_k(t) \) is the allocation size for program \( k \), at time \( t \), and hence
\[ P = \sum_{k=1}^{N} c_k(q_k(t)), \quad 0 = \sum_{k=1}^{N} q_k(t) - M \tag{4.1.3} \]

are the performance measure and the constraint equation, respectively.

The augmented function, \( f_a \), is given by

\[ f_a = P + \lambda \left[ \sum_{k=1}^{N} q_k(t) - M \right] . \]

We assume \( q_k(t) \) need not be integral, as is common; the solutions obtained must therefore be rounded. Equation (4.1.2) yields

\[ \frac{\partial P}{\partial q_k} + \lambda = 0, \text{ for } k = 1, 2, \ldots, N, \tag{4.1.4} \]

which together with the constraint can be solved for the optimal \( q_k(t) \)'s, i.e. \( q_k^*(t) \)'s.

We consider here an example for a cost function of type (A), considered earlier. Assume a Gaussian density model for the program locality size (cf. Section 4.2), where the variance is fixed but the mean over each time interval (quantum, \( \tau \), between allocation decisions) is not. Define the cost as the probability that the "current" locality size exceeds the allocated partition size, i.e.

\[ c_k(q_k(t)) = \int_{q_k(t)}^{\infty} \frac{1}{\sqrt{2\pi}\sigma_k^2} e^{-\frac{(x - \mu_k(t))^2}{2\sigma_k^2}} dx \]

\[ = \text{Prob}[x > q_k(t)] \]

where \( q_k(t) \) = partition size for program k in the time interval \((t, t+\tau)\),

\[ \mu_k(t) \text{ = mean locality size of program k in the time interval} \]

\[ (t, t+\tau), \]

\[ \sigma_k^2 \text{ = variance in locality size of program k}, \]
\( x \) = locality size, a random variable.

This cost function satisfies the continuity conditions. Thus from (4.1.4) we have, \( \partial c_k(q_k(t))/\partial q_k(t) + \lambda = 0, \forall k, \) so

\[
\lambda = (2\pi \sigma_k^2)^{-1/2} \exp\left(-\frac{(q_k(t) - \mu_k(t))^2}{2\sigma_k^2}\right), \text{ for } k = 1, 2, \ldots, N.
\] (4.1.5)

Let \( g_k(t) = (q_k(t) - \mu_k(t))/\sigma_k. \) Then after simplifying, (4.1.5) yields

\[
g_i^2(t) = g_j^2(t) - 2 \ln(\sigma_i/\sigma_j), \text{ for } i, j = 1, 2, \ldots, N. \tag{4.1.6}
\]

Solving this, using the definition of \( g_k \) and the constraint, we obtain,

\[
q_k^*(t) = \mu_k(t) + \sigma_k \frac{1}{\sqrt{N}} \left[ N \sum_{l=1}^{N} \left( \frac{q_l(t) - \mu_l(t)}{\sigma_l} \right)^2 + N \sum_{l=1}^{N} \ln(\sigma_l^2/\sigma_k^2) \right]^{-1/2}
\] (4.1.7)

The optimal partition size for program \( k, \) \( q_k^*(t), \) thus equals the mean locality size, \( \mu_k(t), \) plus a fraction proportional to the standard deviation for program \( k. \) The second term of the expression in (4.1.7) may be simplified by making an assumption on the variance in locality size for programs competing for the main memory resources. In particular, if we assume

\[
\sigma_i^2 = \sigma_j^2 \text{ for all } i, j,
\]

i.e. the variance for all programs is the same, then the optimal partition size becomes

\[
q_k^*(t) = \mu_k(t) + \frac{1}{N} \left[ M - \sum_{l=1}^{N} \mu_l(t) \right]
\] (4.1.8)

The optimal partition size here equals the mean locality size, plus an equal fraction of the 'excess' memory, \( M - \sum_{l=1}^{N} \mu_l(t). \)
Assumption (iii) on the second derivative requires that

\[ q_k^*(t) > \mu_k(t), \quad \text{i.e.} \quad M - \sum_{l=1}^{N} \mu_l(t) > 0, \quad (4.1.9) \]

be true for optimal partitions. This imposes an upper bound on N, the number of programs in main memory (i.e. the degree of multiprogramming) for an optimal partitioning.

In the above discussion, it should be noted that in the limit, if \( N \sum_{l=1}^{N} \mu_l(t) = M \), then the optimal partition sizes equal the respective mean locality sizes which conforms to our expectations. On the other hand, if (4.1.9) holds, as will generally be the case, then our result (4.1.8) shows exactly how the excess memory should be allocated, for programs with identical variances. For unequal variances, which is commonly true for real programs, the excess memory should be allocated according to (4.1.7). This is the main result of this section.

For the equal variance assumption, expression (4.1.9) shows what needs to be done to avoid 'thrashing'. If (4.1.9) does not hold, the degree of multiprogramming should be reduced to prevent thrashing. If in (4.1.9) the excess memory is greater than mean locality size of a program waiting to be loaded into main memory, then the scheduler may decide to load that process.

For the general case of unequal variances, similar results can be obtained. In particular the requirement for avoiding thrashing is,

\[ q_k^*(t) > \mu_k(t), \]

\[ \text{i.e.} \quad \sum_{l=1}^{N} \left[ \frac{(q_l(t) - \mu_l(t))^2}{\sigma_l^2} + \ln \frac{\sigma_l^2}{\sigma_k^2} \right] > 0 \quad (4.1.10) \]
The result, $q^*_k(t) > \mu_k(t)$, to prevent thrashing has been supported through experiments by Rodriguez-Rosell and Dupuy [34].

Next we consider the application of Bayesian decision theory to estimate the mean locality size ($\mu(t)$), as required above.
4.2 ESTIMATION OF LOCALITY SIZE

We consider here the application of Bayesian decision theory to the problem of estimating the mean locality size of an arbitrary program. The actual ("current") locality size, at time $t$ over an interval $(t, t+\tau)$, is the number of distinct pages referenced during that interval. The working set model attempts to estimate this locality size based upon the number of distinct pages referenced during the preceding interval $(t-\tau, t)$. We shall, however, use structural information on the mean and the variance of these sizes (Section 2.4) to provide an a priori estimate, which may then be updated by the sample (reference history) information obtained dynamically during execution over possibly a much longer preceding interval. Regarding the locality size as a random variable, we then address the problem of estimating the locality size "optimally," using the structural a priori information along with the sample information such that the expected value of the "loss" incurred for a wrong estimate is minimized. This loss function should be a function of the difference between the estimate and the actual mean locality size.

The expected value of the loss is dependent on a probability distribution determined adaptively by combining sampled information with that already known. While the parameters of the distribution (e.g. its moments) are considered unknown, and to be determined using a priori and sample information, the form of the distribution must be considered known. A common assumption, for example, is that the locality size is Gaussian [32,55]. While much effort has been spent on justifying or rationalizing this assumption, primarily for tractability reasons, we
recognize that there is no one true or best model for all programs.

Sample information is assumed to be taken from a specified type of process (e.g. Gaussian) with unknown parameters. This information may consist of page reference counts, fault rates, or the like. Although samples may be taken "continuously" by program monitoring, only at discrete time instants are they used to obtain updated information. We distinguish between the times that samples are taken ("observation instants") and the times that the probability distribution is updated ("estimation instants"). The latter are chosen to coincide with the times that the estimates are needed—e.g. when partition size reallocations are made.

Most locality size estimation algorithms rely solely on sample or usage information (LRU, FIFO, etc.) to the exclusion of a priori information, in part because of the unreliability of a priori (statistical) information for pre-loading purposes. This objection, however, is not valid in our context since in a Bayesian scheme, a priori information is continually updated so that initial estimates need not be highly accurate. The a priori information we propose to use is that found in program page graphs, which can easily be provided by the system "paginator" (i.e. that routine which segments programs into pages in the first place). The precise information of interest is the strongly connected subgraphs containing each page.

In the following sections, we first review the basic theoretical concepts upon which our estimation procedure is based. Subsequently, we discuss loss functions, prior distributions, likelihood functions, and posterior distributions in the context of locality size. We conclude
with an example illustrating these ideas, using a Gaussian model.

4.2.1 Bayesian Decision Theory [37]

Bayesian statistics provides a method of combining new information with previously available information so as to obtain better estimates of unknown parameters. Suppose that the uncertain quantity (or parameter) of interest is a random variable \( \theta \). Suppose further that the information concerning \( \theta \) is summarized by sample statistics \( y \) (information obtained from the process) and a prior distribution on \( \theta \) (prior information). Then, the posterior distribution \( f(\theta | y) = f(\theta = \theta | y = y) \) may be determined using the likelihood function \( f(y | \theta) = f(y = y | \theta = \theta) \) and the prior distribution \( f(\theta) = f(\theta = \theta) \). The mechanism employed is Bayes' Theorem:

\[
f(\theta | y) = \frac{f(\theta, y)}{f(y)} = \frac{f(\theta) f(y | \theta)}{\int_{-\infty}^{\infty} f(\theta) f(y | \theta) d\theta}
\]

(4.2.1)

It is assumed here that \( \theta \) is a continuous random variable. (A discrete version of Bayes' Theorem would be used otherwise.) The sample statistics may be discrete or continuous. The posterior distribution then summarizes the information concerning \( \theta \), utilizing the sample outcomes \( y \) as well as the prior information on \( \theta \).

It is important to determine the likelihood function and the prior distribution appropriately. The likelihood function is uniquely determined once certain assumptions about the process being sampled are made, e.g. that the process obeys a Gaussian distribution. For prior distributions, however, some simplifying assumptions are generally needed so
that a solution to Equation (4.2.1) will be mathematically tractable. This requires choosing the prior distribution from a family of distributions known as the "conjugate priors" for the given likelihood function. For a Bernoulli likelihood function, the conjugate prior is a beta distribution; for a Gaussian function, the conjugate prior is Gaussian. For a conjugate prior distribution the posterior distribution belongs to the same family as the prior, which is the motivation for its choice. The values of the parameters of the prior distribution are chosen to reflect the a priori information available.

To obtain an optimum estimate of \( \theta \), the "average loss" obtained from the posterior distribution and a loss function is minimized. A loss function gives the consequences of making an incorrect decision, if a particular value of \( \theta \) is chosen. Let \( \theta^* = \) the optimum value selected, and \( \theta = \) the actual value of parameter \( \theta \). Then the average loss, \( \bar{L}(\theta^*) \),

\[
\bar{L}(\theta^*) = \int_{-\infty}^{\infty} L(\theta,\theta^*)f(\theta)d\theta
\]

(4.2.2)

where \( L(\theta,\theta^*) \) is the loss function and \( f(\theta) \) is the posterior distribution. Optimally \( \theta^* \) should be chosen equal to \( \theta \); if \( \theta^* < \theta \) or \( \theta^* > \theta \), losses are incurred, represented by the loss function \( L(\theta,\theta^*) \).

The Loss Function:

We are interested in the estimation of mean locality sizes. Let \( \theta \) represent the actual mean locality size and \( \theta^* \) its estimate. Then for \( \theta^* < \theta \), the loss function should in some way represent the excessive overhead due to page faults; for \( \theta^* > \theta \), the loss function should be a measure of the inefficient use of main memory and the ensuing loss in
degree of multiprogramming and system response time. The loss function for both \( \theta^* > \theta \) and \( \theta^* < \theta \) should be representable in like units so that a mathematical optimum can be obtained. We adopt the following loss function:

\[
L(\theta, \theta^*) = \begin{cases} 
C_1 \text{Pr}\{\theta^* > \theta\} & \text{if } \theta^* > \theta \\
C_2 \text{Pr}\{\theta^* < \theta\} & \text{if } \theta^* < \theta 
\end{cases}
\]  

(4.2.3)

where \( C_1 \) and \( C_2 \) are proportionality constants to convert the probabilities to the required loss units. (\( \theta^* \) will turn out to be independent of \( C_1 \) and \( C_2 \) for this loss function.) Substituting Equation (4.2.3) in Equation (4.2.2), and assuming a specific distribution for the mean locality size, the optimum estimate for \( \theta \) can be found in theory by solving \( \frac{dL}{d\theta} = 0 \). In this way, we obtain the optimum estimate

\[
\theta^* = E(\tilde{\theta}) = \text{expected value of } \tilde{\theta}.
\]  

(4.2.4)

This estimate is also optimum for quadratic loss functions

\[
L(\theta, \theta^*) = k(\theta - \theta^*)^2.
\]

In summary, the optimum estimate of the mean locality size parameter, relative to the loss function in (4.2.3), is equal to its expected value evaluated according to some given probability distribution. This distribution can be revised based upon sample information to yield better estimates using (4.2.1) iteratively as we shall see.
Prior Distributions:

The prior distributions for the mean locality size parameter should be chosen to reflect a priori information. Its form is dictated by other considerations (it should be a conjugate prior, as discussed previously), so it remains to determine its distribution parameters. To this end, we utilize here program structural information, as is available in program page graphs. The basic idea is this: the "localities" to which each page can belong correspond to the strongly connected subgraphs, as discussed previously in Chapter 2. The parameters (moments) of the prior distribution, \( f(\theta) \), may then be estimated by the mean, variance, etc., of the sizes of localities of the program. Thus, we may let

\[
\mu = \text{a priori mean} = \frac{1}{n} \sum_{i=1}^{n} \lambda_i,
\]

\[
\sigma^2 = \text{a priori variance} = \frac{1}{n} \sum_{i=1}^{n} (\mu - \lambda_i)^2,
\]

where \( \{\lambda_i\} \) are the mean sizes of the "localities" and \( n \) their number. While the above scheme is simple, it is not likely to be very helpful since programs cannot be expected to execute in a "global" (overall) average mode. If we assume the "locality" model, then the programs are expected to execute in a "local" mode. Then the mean and variance to be used should be those corresponding to the current locality, i.e. the locality in which the program is currently executing. These values (of mean and variance) then define the prior distribution for the program at the current time, and remain the same so long as the process
remains in the same locality. As soon as it is determined that the process has entered a new locality, the mean and variance values must be changed correspondingly.

One means of implementing the above requires keeping lists of pages belonging to each locality and their corresponding means and variances (in the mean locality size parameter) on the same page table as the page mapping information. Also for each page, we associate an "entry" bit which is used to determine when a process is entering a new locality. The manner of determination of localities and their means and variances was described earlier in Chapter 2.

Likelihood Functions:

The likelihood function, \( f(y|\theta) \), is obtained by a combination of the sample information and an assumed process model. We consider below two types of process models (Bernoulli and Gaussian) and their associated likelihood functions.

Case I (Bernoulli). Let the execution of each instruction be regarded as an event. Define a random variable \( x_j \) as follows:

\[
\tilde{x}_j = \begin{cases} 
1, & \text{if the } j\text{-th instruction refers to a distinct page} \\
0, & \text{otherwise.} 
\end{cases}
\]

By a "distinct" page, we mean one not previously referenced since time \( t \), the beginning of an interval of time \( (t,t+t) \). In this interval, the sample information is \( y = \{n,r\} \), where
n = total number of instructions executed,
\[ r = \sum_{j=1}^{n} x_j \] = total number of distinct pages referenced.

The process may be modeled as a Bernoulli process if we assume for each interval that the samples are independent and the probability, p (unknown), of referencing a distinct page, on each instruction execution, is constant during the interval. The density function of the process is given by

\[
f_B(r|n,p) = \binom{n}{r} p^r (1 - p)^{n-r},
\]

where \( r \) is the expected number of distinct page references in \( n \) executions. The likelihood function for the Bernoulli process with observation \( y = \{n,r\} \) is then given by

\[
f(y|\theta) = f(y|p) = \binom{n}{r} p^r (1 - p)^{n-r} \tag{4.2.5}
\]

**Case II (Gaussian).** Let \( x \) denote the number of distinct pages referenced in a given time interval. Let us assume that \( x(t) \) is a stationary Gaussian process with mean \( \mu \) (to be estimated) and variance \( \sigma^2 \) (given a priori). Then the density function is given by

\[
f(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)
\]

Let \( x_i \) = number of distinct pages referenced in the \( i \)-th observation interval where the \( x_i \) are independent and identically distributed. Then let the sample information \( y = \{x_1, x_2, \ldots, x_n\} \) be represented by
\{n,m\}, where

\[ n = \text{number of observation intervals, and} \]

\[ m = \text{sample mean} = \frac{1}{n} \sum_{i=1}^{n} x_i, \]

since only \(n\) and \(m\) explicitly appear in the expression for the likelihood function given below. The likelihood function for observations \(y = \{n,m\}\) is then

\[ f(y|\mu) = \prod_{i=1}^{n} f(x_i|\mu,\sigma^2) = k_m \exp\left(-n(m - \mu)^2/2\sigma^2\right) \quad (4.2.6) \]

where \(k_m\) is a constant with respect to \(\mu\) and may be determined using the property \(\int_{-\infty}^{\infty} f(y|\mu)dy = 1.\)

**Posterior Distributions:**

Given a prior distribution and a likelihood function, the posterior distribution, \(f(\theta|y)\), can be calculated using Equation (4.2.1). This distribution represents the total information available about the uncertain parameter at any given time. The effect of adding sample information to a prior distribution is to make the distribution more peaked, so that the determination of the parameter becomes less uncertain. The posterior distribution may in turn be regarded as a prior distribution, which when combined with further sample information, yields another posterior distribution, and this process may then be iterated. Recalling that posterior distributions are used to evaluate
expected losses associated with estimates of mean locality sizes, we thus have an adaptive method for estimating mean locality sizes which uses both prior and sample information.

**Case I** (Bernoulli). If the likelihood function is given by Equation (4.2.5), and the prior distribution is a beta distribution $\beta(n', r', p)$, then the posterior distribution for a sample $y = \{n, r\}$ is a beta distribution $\beta(n'+n, r'+r, p)$.

**Case II** (Gaussian). If the likelihood function is given by Equation (4.2.6), and the prior distribution is a Gaussian distribution with mean $m'$ and variance $\sigma'^2$, then the posterior distribution for a sample $y = \{n, m\}$ is a Gaussian distribution with mean $(n'm'+nm)/(n'+n)$ and variance $\sigma^2/(n'+n)$, where $n' = \sigma^2/\sigma'^2$.

### 4.2.2 Bayesian Estimate of Locality Size

We present here a method of estimating the mean locality size, assuming a Gaussian density model for the process, and utilizing the ideas developed in the previous section. Let $\mu$, a random variable, denote the number of distinct pages referenced in an interval $(t, t+\tau)$. Then our objective is to determine an optimal estimate of $\mu$, $\mu^*$, such that the loss function (4.2.3) is minimized.

Let $\bar{\mu}$ be the mean (the unknown parameter) of the Gaussian distribution assumed for the process, and $y = \{n, m\}$ be the observation set, where $m$ is the average number of distinct pages referenced in each of the $n$ estimation (execution) intervals of the present locality. The
likelihood function is, for the above assumptions,
\[ f(y|\mu=\mu,\sigma^2) = k_m \exp\{-n/(2\sigma^2)(m-\mu)^2\} \]
(4.2.7)
The form of the (conjugate) prior density function, i.e., the distribution on the mean locality size, is also Gaussian,
\[ f(\mu) = (1/2\pi\sigma'^2)^{1/2} \exp\{-(\mu-m')^2/2\sigma'^2\} \]
(4.2.8)
where \( m' \) is the a priori mean locality size, and \( \sigma'^2 \) is the a priori variance. (The parameters \( m' \) and \( \sigma'^2 \) are determined for the current locality using structural considerations as in Section 2.4.) The posterior density function is given by
\[ f(\mu|y) = k_m \exp\{-(\mu-\mu'')^2/2\sigma''^2\} \]
(4.2.9)
where \( k_m \) is determined using \( \int_{-\infty}^{\infty} f(\mu|y) d\mu = 1 \), and
\[ \mu'' = \frac{n'm' + nm}{n'^2 + n} , \quad \sigma''^2 = \frac{\sigma^2}{n'^2 + n} , \quad \text{and} \quad n' = \frac{\sigma^2}{\sigma'^2} \]
(4.2.10)
The posterior distribution for the mean locality includes observed (historical) information \( y \) to update the a priori parameters \( m' \) (and \( \sigma'^2 \)).

The optimum value of \( \tilde{\mu}, \mu^* \), i.e., the optimal estimate of the mean locality size to the mean of the posterior distribution \( \mu'' \), by (4.2.4), hence
\[ \mu^* = \left( \frac{n'}{n'^2 + n} \right) \times \left\{ \text{a priori estimate of} \right\} \]
\[ \text{mean locality size} \]
\[ + \left( \frac{1}{n'^2 + n} \right) \times \left\{ \text{sum of the number of distinct pages referenced in each of the n observation intervals} \right\} \]
The Bayesian estimate of mean locality size thus uses the history of program execution, in the current locality, and the a priori estimate. We show next, through simulation experiments that by using Bayesian estimates of the mean locality size, the total number of page faults
produced, and hence the performance of the system, is improved considerably.

4.2.3 Simulation Model

To illustrate the effectiveness of the Bayesian estimation procedure, we present the results of simulation experiments. We resorted to simulations because to include the various characteristics of the Bayesian algorithm and dynamic partitioning would make the mathematical model too complex for simple analysis.

In our simulations, the "locality" behavior of programs is modeled for generation of page reference strings. In addition we also used a page trace generated for an actual run of the program. The results obtained from both methods were comparable. First we will describe the simulation model.

Description of Simulation Model

The simulation consists of the following:

- Program behavior and locality model
- Page reference string generation
- Modeling of various partitioning schemes
- Performance criteria
- Modeling of the replacement algorithm

Program behavior is modeled based upon Denning's locality model as described in Section 1.1. A program is assumed to consist of a number of localities and each locality is assumed to maintain control for a period of time before a transition is made to another locality.
The time spent in each locality on subsequent executions may be different.

This is modeled by determining locality duration probabilistically. A locality duration, $T_{\lambda_i}$, for locality $\lambda_i$, is a random variable which is assumed to be a multiple of the Min_loc_time interval. The Min_loc_time is the minimum duration spent in any locality. At the end of each such interval, there is a certain probability of making a transition to another locality. This is illustrated in Figure 4.2.1.

We assume that:

- Control may stay in a locality for an integral multiple of the Min_loc_time interval.
- At the end of each such interval there is a given probability of making a transition to another locality.
- For each locality, $\lambda_i$, the probability of transition, $P_{\lambda_i}$, is constant with a value dependent on the locality.
- The next locality chosen on a transition may be decided probabilistically, or the sequence may be predetermined.

The average locality duration (for $\lambda_i$) is then given by,

$$T_{\lambda_i} = \frac{1}{P_{\lambda_i}} \times \text{Min}_\text{loc}_\text{time} \quad (4.2.12)$$

(Min_loc_time and $\{P_{\lambda_i}\}$ are parameters for the simulation.)

Each locality itself is modeled using Coffman and Ryan's mathematical model of locality [32]. They assume that a locality can be modeled by a Gaussian process and each locality is defined by a mean locality size and a variance (in locality size). Distinct localities may thus be defined by distinct sets {mean, variance}. A transition
from one locality to another only indicates a change in the values of mean and the variance, and does not necessarily indicate a total change of pages belonging to the locality. The pages among subsequent localities may thus be overlapping. The mean and variance are used as the a priori parameters for the estimation of locality size. On a change of locality, the a priori parameters used correspond to the new locality. The overlap among localities is a parameter in the simulation.

Page Reference Strings

Sequences of page references, r₁, r₂, ..., rᵢ, ..., (where each page reference, rᵢ, denotes a page taken from a set of N' pages), can be generated using stationary probability distributions, \{bₖ\}_{k=1}^{N'} on LRU stack positions. This is based on the assumption that successive stack distances are statistically independent and identically distributed random variables. If

\[ sᵢ = \{sᵢ(1), sᵢ(2), ..., sᵢ(k), ..., sᵢ(N')\}, \]

denotes the LRU stack after the \( i \)th page reference, \( rᵢ \), and \( sᵢ(k) \) the \( k \)th most recently referenced page relative to \( rᵢ \), then for all reference strings and for all \( i \),

\[ \Pr(rᵢ = sᵢ₋₁(k)) = bₖ, \quad 1 < k \leq N'. \]

The property of locality implies that \( b₁, b₂, ..., bₙ \), is a non-increasing sequence. In our simulations, we used Uniform and Geometric distributions on \( \{bₖ\} \) for generation of reference strings. Coffman and Ryan [32] found that this model for page reference strings justifies the Gaussian model assumed for the localities. (This is similar to the simple LRU model of Spirn, Denning, and Savage [7].)
The stack distance probabilities, \( \{b_i\} \), are different for the different localities and as soon as the locality change occurs, these probabilities are changed to correspond to those for the new locality.

**Modeling of Partitioning Algorithms**

*Fixed partitioning* is simple to simulate. All processes in memory are allocated equal amounts of physical memory, \( MM/DM \), where

\[
MM = \text{Physical memory size in pages},
\]
\[
DM = \text{Degree of Multiprogramming}
\]

Any reference with stack distance greater than \( MM/DM \) causes a page fault.

*Dynamic partitioning* is simulated by measuring the working set sizes (WSS) for a fixed \( T \) (the window size). The WSS is an estimate of locality size as per Denning's working set principle. The memory allocated to each process is in proportion to its WSS. The WSSs are measured at the end of fixed intervals \((=T)\), and reallocations are made (partition sizes re-determined) at the same time. This allows a meaningful comparison between the dynamic and Bayesian partitioning. The relation between reallocation times and locality duration are shown in Figure 4.2.2.

*Bayesian partitioning* is similar to dynamic partitioning except

- the locality size is estimated using Bayesian statistics, as discussed previously,
- at the beginning of a new locality the a priori parameters (used for Bayesian estimation) correspond to those for the new locality.

Figure 4.2.3 shows the relation among various times and intervals for
Figure 4.2.1 - Timing Diagram

Figure 4.2.2 - Dynamic

Figure 4.2.3 - Bayesian
Bayesian partitioning. At reallocation instants the locality size is re-estimated for each process using a priori parameters and the sample information. The sample information used is the number of pages referenced in each of the reallocation intervals since the beginning of the present locality. Let
\[ m_i' = \text{mean locality size of the present locality (ith locality)}, \]
\[ \sigma_i'^2 = \text{variance in locality size for the present locality} \]
These a priori parameters are determined by simulating each locality separately, by generating the page trace for corresponding values of \( \{b_i\} \), and by estimating its mean and the variance in locality size for a window size \( \tau \) equal to the reallocation interval. The locality size is estimated using Denning's working set principle for this purpose.

The locality sizes are thus estimated for a reallocation interval and updated at the end of each such interval, as for the dynamic partitioning approach. In both approaches the estimates of locality size depend upon the number of distinct pages referenced in a reallocation interval. In the Bayesian case, however, these values are averaged over a number of such intervals.

The overall process variance, \( \sigma^2 \), is estimated by assuming all localities to be equally probable. (If the respective weights were known, this information could be used instead.) Let
\[ m_j = \text{number of distinct pages referenced in jth reallocation interval of current locality}, \]
\[ n = \text{total number of reallocation intervals in current locality}. \]
Then the Bayesian estimate of locality size (BLS) is given by (4.2.11),
The a priori parameters \((m_i^1\text{ and } \sigma_i^{12})\) remain the same during each locality. On a change of locality these are changed to correspond to those for the new locality, and \(n\) is set to zero (previous sample information is discarded).

**Performance Criteria**

Memory exceedance and page fault counts are used as performance indicators. Memory exceedance (M.E.) for various partitioning algorithms is given by (cf. Coffman and Ryan [32])

\[
\text{Fixed: } M.E. = \frac{DM \sum_{i=1}^{DM} (WSS_i - MM/DM)}{DM} \text{ summed over each } i \text{ such that } WSS_i > MM/DM
\]

\[
\text{Dynamic: } M.E. = \begin{cases} 
DM \sum_{i=1}^{DM} WSS_i - MM, & \text{if } \sum_{i=1}^{DM} WSS_i > MM \\
0, & \text{otherwise}
\end{cases}
\]

\[
\text{Bayesian: } M.E. = \begin{cases} 
DM \sum_{i=1}^{DM} BLS_i - MM, & \text{if } \sum_{i=1}^{DM} BLS_i > MM \\
0, & \text{otherwise}
\end{cases}
\]

**Replacement Algorithm**

The replacement algorithm modeled was LRU. Other algorithms could have been used but LRU was particularly appropriate since the page traces were generated using an LRU stack.
4.2.4 Simulation Results

The simulations were performed for a fixed degree of multiprogramming and by partitioning the main memory among competing processes in the ratio of their locality size estimates (except for fixed partitioning where all processes were allocated equal amounts of main memory). Two criteria for performance measurements were considered, that of memory exceedance, and the total number of page faults generated. These two criteria were evaluated for various page traces, using different types of distributions on stack distances, and for different main memory sizes. The results obtained, for both criteria, indicated the superiority of the Bayesian estimation scheme. The graphs for memory exceedance are shown in Figures 4.2.4 to 4.2.7. The values of various parameters are indicated therein. Note that in Figure 4.2.4, the mean and variance parameters for the overall process are the same as the ones for the example of Coffman and Ryan (Figure 4, [32]). The shapes of the curves obtained for fixed and dynamic are almost identical to the ones obtained by them analytically. This may be regarded as a validation of the simulation technique employed here. The page fault curves for the three algorithms are plotted in Figures 4.2.8 to 4.2.11. Here, as for memory exceedance, the improvements produced by Bayesian techniques are significant, especially for medium values of partition sizes. This is to be expected since if the memory available is too small (much smaller than the average locality size), the processes would page fault quite often regardless of which partitioning algorithm is used, and their performances would not be too different. Similarly for large memory sizes, the partition sizes are greater than the actual
Probabilistic Data
Degree of Multiprog. = 9
Window Size = 75
WSS: mean = 10, \(\sigma = 3\)
LRU Stack Dist.: Uniform

Figure 4.2.4 - Mem. Exc. vs Mem. Size

Figure 4.2.8 - Page Fault vs Mem. Size
Probabilistic Data
Degree of Multiprog. = 9
Window Size = 50
WSS: mean = 13, $\sigma = 2.5$
LRU Stack Dist.: Uniform

Figure 4.2.5 - Mem. Exc. vs Mem. Size

Figure 4.2.9 - Page Fault vs Mem. Size
Probabilistic Data
Degree of Multiprog. = 9
Window Size = 50
WSS: mean = 10, σ = 2.2
LRU Stack Distr.: Geometric

Figure 4.2.6 - Mem. Exc. vs Mem. Size

Figure 4.2.10 - Page Fault vs Mem. Size
Hatfield's Data
Degree of Multiprog. = 6
Program Size = 232 pages
Window Size = 50

Figure 4.2.7 - Mem. Exc. vs Mem. Size

Figure 4.2.11 - Page Fault vs Mem. Size
Performance for Varying Locality Duration
Degree of Multiprogram. = 9
Memory Size = 99 pages
Window Size = 50
LRU Stack Dist.: Uniform

Figure 4.2.12 - Mem. Exc. vs Mem. Size

Figure 4.2.13 - Page Fault vs Mem. Size
memory requirement, so again the performances of the algorithms should not be too different. The main effect of the dynamic and Bayesian algorithms is felt when the main memory available is in the medium range. Due to its more accurate estimation of locality size, the performance of the Bayesian algorithm should be much superior.

The above results are all for the case in which the locality duration is larger than the reallocation interval. Since the performance of the dynamic algorithm depends very much on the value of this parameter, namely the average locality duration, we also compared results for varying average locality duration over a wide range of values. First let us discuss what we expect for the performance of the dynamic and Bayesian estimation schemes for varying average locality duration. In discussing the performance (and in the simulation) we assume that as soon as a locality change occurs, the partition sizes for the Bayesian algorithm are readjusted. We note also that degree of multiprogramming is assumed fixed and available memory is allocated in proportion to the estimated locality sizes of programs.

For average locality duration < reallocation interval, there may be one or more locality changes in each interval. Thus the number of distinct pages referenced in an interval may be quite large and hence the locality size estimate for the dynamic algorithm will be inflated and not correctly represent the requirements of the program. The memory exceedance associated with the dynamic algorithm would then be large (close to that for the fixed algorithm) since locality size estimates would be more likely to exceed available memory. For the Bayesian algorithm, however, the memory exceedance would not be as large
because the estimated locality sizes (utilizing a priori information) are much smaller and follow the program needs more closely. The number of page faults for the dynamic algorithm would also be larger than those for the Bayesian algorithm because inflated locality size estimates for the dynamic algorithm would result in partition sizes being almost fixed for the programs in memory even though their current requirements may be quite different.

For average locality duration \( \geq \) reallocation interval (but not too great), the Bayesian algorithm would perform better both in memory exceedance and total number of page faults, because of the savings effected though correct estimation of locality sizes at locality changes. The simulation results cited earlier were all in this region and showed considerable improvements for the Bayesian algorithm. For very large values of average locality duration, the dynamic and Bayesian performances should get closer, since the number of locality changes, which cause the major degradation in dynamic algorithm performance, are much less in number. However, the Bayesian algorithm should still perform better, even for zero locality changes, because of a better initial (and subsequent) estimation of locality sizes.

Figures 4.2.12 and 4.2.13 give the results of simulating the varying average locality duration. The performance of the dynamic and Bayesian algorithms was as expected. The Bayesian algorithm performs better over all values of average locality duration.

In summary, Bayesian estimation differs from dynamic estimation in two respects; namely, the use of a priori estimate at changes of locality, and the use of an average estimate for locality size within a
locality. The advantage of the first condition is obvious (further discussed in Section 4.3). The second difference may not lead to a better estimation if within a locality the locality size changes rapidly. However, within a locality the locality size is not generally expected to exhibit such behavior. This is substantiated by the simulation results presented here and by definition of locality behavior [6,7,32].
4.3 ESTIMATION OF LOCALITY MEMBERSHIP (LOADING POLICY)

Estimation of membership is the determination of what pages to load as members of a locality, and what page(s) to replace or unload when more pages are desired. This problem may be considered from two points of view:

- Demand Paging
- Pre-loading

We first discuss these two approaches to page loading. For the purposes of the following discussion, we assume a system with a CPU, main memory, a drum as secondary memory, and a separate memory management processor.

4.3.1 Demand Paging

One approach used for loading pages into main memory is upon demand, i.e. pages are only loaded when actually required. At the beginning of program execution (and also at the beginning of a quantum for time-shared systems) the page which is currently active is loaded and remaining pages are loaded as needed. By loading pages as required, the total number of load operations are minimized (for a particular replacement algorithm and page allotment). However, lower page fault rates require larger resident set sizes. Furthermore, each time a page fault occurs, the system must switch to another process whose resident set is in main memory to keep the system utilization high. Thus a large number of such processes will have to be maintained in main memory because while a process' locality is being constructed the process page faults very often. The amount of main memory required to
keep the processor busy would be quite large.

4.3.2 Pre-loading

Pre-loading has been designed to overcome the shortcomings in demand paging systems outlined above. Consequently, pre-loading is best suited for time-shared systems where a process is swapped in at the beginning of its quantum and swapped out after it has executed its quantum. In pre-loading the entire locality is loaded as a batch before a process is given control of a processor. The main memory is no longer tied up as long as for demand paging, because the pages of the locality are loaded as a batch and not one-by-one on demand. For a dynamically allocated drum (cf. Greenberg [41]) normally used for pre-loading systems, this time is much smaller, approximately equal to one drum revolution time. Memory utilization is thus much better. Greenberg shows that for the same degree of performance (for a time-shared environment), pre-loading has lower main memory requirement as compared to demand paging. This is attributed mainly to the amount of main memory saved for processes waiting for one or more of their pages to be loaded, since a lower number of such processes will be required.

In pure pre-loading, on a page fault the entire resident set is swapped out onto the drum and the new pre-load set is estimated before the process is scheduled again. The pre-load set should be chosen such that the process executes for a significant interval (quantum time) before page faulting. Thus for pure pre-loading a very good estimate of the locality is required. This in general is difficult to obtain because of the probabilistic nature of program execution. The historical
estimates of localities rarely describe the imminent needs of a process with complete accuracy. Therefore, the pre-load set, in general, would contain a large number of pages which are not subsequently referenced, to minimize the page fault probability. The average pre-load set would be much larger than the average locality size for demand paging.

The scheme used in most systems is therefore a pre-loading scheme which operates between the two extremes in page loading policy. It pre-loads an estimate of the locality and allows a certain amount of page fault activity during the quantum. In such schemes, at the start of a quantum the pre-load set (determined as per criteria to be discussed subsequently) is loaded into the main memory. Once the pre-load set is in main memory the treatment of the process is the same as in demand paging. Any page faults during the quantum are serviced by replacing a page. (In Section 4.4.3, we shall show that the number of page faults produced will be much smaller as compared to demand paging; also, the number of process swapping operations will be smaller than those for pure pre-loading.) We perform below an approximate analysis to show that pre-loading has a smaller cost compared to demand paging for time-shared systems, measured in terms of the time spent for loading operations. (We neglect delays due to queueing in the loading cost.)

Let 
\[ c_p = \text{cost of pre-loading} \]
\[ c_d = \text{cost of demand paging} \]
\[ p = \text{average pre-load set size} \]
\[ D = \text{average number of pages demanded during quantum by demand paging} \]
\[ d = \text{average number of pages demanded during quantum for} \]
pre-loading

\[ t_r = \text{drum revolution time} \]

\[ t_r/2 = \text{average load time for a demanded page} \]

\[ n \cdot t_r = \text{average load time for a pre-load set} \]

Then

\[ c_p = n \cdot t_r + d \cdot \frac{t_r}{2} = \left\{ \text{time to load} \right\}_{\text{pre-load set}} + \left\{ \text{time to load} \right\}_{\text{demanded pages}} \]

\[ c_d = D \cdot \frac{t_r}{2} = \text{time to load demanded pages} \]

Hence

\[ c_d - c_p = \frac{t_r}{2} \left\{ D - (d + 2n) \right\} \]

Thus if the pre-load set contains \( 2n \) or more pages which are the same as the ones in the working set obtained through demanded paging, we have

\[ c_d \geq c_p, \quad \text{if} \quad D \geq d + 2n \quad (4.3.1) \]

In other words the pre-load set estimate should be such that it saves at least \( 2n \) page faults on the average, to yield an improvement over demand paging. In general, however, the pre-load set would do much better than this, i.e.

\[ c_d > c_p , \]

since \( n \) is usually small \((\leq 2)\), as we now argue. (The importance of determination of the pre-load set membership is discussed in the following section.)

An efficient dynamic drum policy requires that sector conflict be minimized, i.e. no two pages of the same locality be written at the same sector position. Thus a page being written out is restricted to those sectors which do not contain another page of the same locality.
Here it is assumed that the maximum locality size is smaller than the number of sectors $S$, on the drum. It is then possible to load the entire locality in one revolution time. This optimum reading time may be exceeded due to insufficiency of free memory, into which the pre-load set is to be loaded, or by sector interference due to loading of pages belonging to other processes. The number of pages to read in, on any sector queue, cannot exceed the number of processes in a loading state. (For convenience we assume that there are no shared pages.)

On the average if there are $m_1$ processes causing sector interference, due to $m$ average number of processes in a loading state at any time, then the average number of drum revolutions, $n$, required to load the entire locality of a process is given by $n \leq \sum_{i=1}^{m_1} i/m_1$. (This is because, of the $m_1$ loading processes on the same sector, the highest priority process is read within one revolution and only the lowest priority process is read in $m_1$ revolutions.) The value of $m_1$ will depend upon the average locality size and the total number of sectors. Since not all loading processes will be loading the entire localities (some may be loading just one demanded page), and if the average locality size $\ll S$, the value of $m_1$ will be much lower than $m$. Thus for a system with an average of 4 processes in a loading state, a reasonable value for $m_1$ is, $m_1 \leq 2$. Then the average number of drum revolutions required to load a set is, $n \leq 1.5$.

4.3.3 Estimation of Membership

The estimation of membership in a demand paging environment is purely a question of deciding what page to replace on demand for a new
page, if the main memory allotted to the program is full. Thus this reduces to the problem of choosing an appropriate replacement algorithm (cf. Chapter 3). The estimation of membership for pre-loading algorithms is a more difficult problem, in that it requires estimation of the future program requirements with more accuracy. For the pre-loading scheme under consideration, there are two aspects to the determination of membership, namely

- determination of pre-load set
- determination of the page to replace on a page fault during a quantum

The second problem is solved by utilizing a proper replacement scheme. The estimation of pre-load set is the more crucial aspect of pre-loading. As we saw in the analysis in the previous section, for efficient operation the pre-load set should be such that $2^n$ or more page faults are saved. If the pre-load set is estimated incorrectly, it will result in a larger number of page faults and the system would degenerate into a demand paging system.

Greenberg in his discussion on pre-loading [41] does not describe how pre-load sets are to be estimated. He refers to Denning's working set theory while considering this. Denning's working set strategy (as well as the BCC CWS strategy to be described in Appendix D) bases the estimate on the history of program execution. This is unsuitable for a pre-loading environment because clearly it cannot anticipate use of pages which have not been recently referenced. Thus if a process is in the midst of changing its 'locality,' the working set will not be a good estimate of the new locality until after all the pages of the new
locality have been referenced. During this period, all estimates of the locality are based on historical information alone and will consequently be erroneous and lead to a number of page faults. Simulation results (given in Section 4.4.3) show that this may lead to significantly poorer performance.

This points to the desirability of a predictive algorithm for determination of pre-load sets. In particular, we contend that structural information is of great importance for this purpose. Structural information can be used to determine the localities in a program (as explained in Section 2.4.1). Then at the beginning of a locality the pages corresponding to it may be loaded as the pre-load set. This would save a large number of page faults depending upon how accurate the a priori estimate of localities are. The Bayesian algorithm as described earlier makes use of structural information for determination of localities, and this seems to be ideally suited for a pre-loading environment.

We describe below a Bayesian locality estimation algorithm for a pre-loading environment.

4.3.4 Bayesian Pre-loading Algorithm

The Bayesian algorithm makes use of a priori locality information to estimate program locality size, as explained earlier. Since we can also determine an a priori estimate of the membership of each locality, as in Chapter 2, each pre-load set can be made equal to the membership estimate associated with the page to be given control. Depending upon how accurate these estimates are, a number of page faults which would
otherwise occur due to changes in locality can be saved by this algorithm. The estimation of pre-load sets is carried out at the following instants:

• start of a new locality
• end of a quantum

The major effect of the Bayesian approach will be due to a change of pre-load set at the start of a new locality. As soon as it is sensed that a process is entering a new locality, the quantum is prematurely terminated and the process is swapped out to the drum. The pages for the next locality are swapped in at the beginning of the next quantum. To avoid loading a locality prematurely, the change of locality should be properly detected. Thus locality numbers should be assigned with great care, to those instruction pages which are the entry pages of localities. The rest of the pages are assigned a locality number of zero, indicating that these pages are not to be used for determination of a change of locality.

We next consider implementation considerations for the pre-loading scheme on the BCC 500.
4.4 IMPLEMENTATION ON BCC 500

In this section we evaluate the feasibility of implementing predictive pre-loading algorithms on a particular computer system, the BCC 500. (The BCC 500 system is a time-shared paged virtual memory multiprocessor computer system designed to support a large number of concurrently active users. We give a brief description of relevant aspects of it in Appendix D.) We also compare the performances of the BCC pre-loading algorithm and a pre-loading scheme based on Denning's working set concept with the predictive schemes introduced here.

4.4.1 Preliminaries

The system load consists of requests for the processor (CPU) and main memory (and perhaps other resources, which we neglect) by a stream of processes. A process is defined as a program in its execution environment. Associated with each process in the system at any given time is a locality or a working set of pages. Each process in the system is given a quantum every so often, during which the process does its computing. For a process to execute, i.e., obtain control of a processor on being scheduled, the entire locality must be in main memory. A process is allowed to execute for the entire quantum allotted to it unless it blocks for I/O or for some other reason, or if it is preempted by the scheduler for running a higher priority process. At the end of the quantum, or on preemption, the pages of main memory assigned to the process are released. All pages which have not been modified are immediately available to the system, while the modified pages have to be
written out onto the auxiliary memory before these can be reallocated.
To keep the CPU occupied, there are a number of processes resident at any time. A process in the system may be in one of the following four states, at any given instant:

- the request state
- the loading state
- the unloading state
- the ready state

A process is in the request state when it is scheduled to be loaded into main memory. A process is in the loading state when its locality (one demanded page or the entire locality, as at the beginning of a quantum) is being loaded. Once the locality is entirely resident the process is in the ready state, ready to be given control of a CPU for execution. Processes in the loading or ready state are also referred to as active processes. When a process completes its quantum or is preempted, the process enters the unloading state. During this state, the process is being swapped out to the drum.

4.4.2 Pre-loading Algorithms

We describe here the pre-loading algorithms which were simulated by us for the purpose of evaluating the historical and predictive information for pre-load set design. The algorithms simulated were the BCC CWS algorithm (described in Appendix D), a Working Set algorithm (for a pre-loading environment), the Bayesian algorithm (described in Section 4.3), and a Predictive BCC algorithm.
Working Set algorithm. Here, the working set at the end of a quantum is treated as an estimate of pre-load set for the next quantum. The pre-load set is then as good an estimate of memory requirement as a working set, in a non-swapping batch environment. The working set is the set of pages referenced in the last $T$ time-interval of execution. The pre-load set estimate is based upon historical information alone.

Predictive BCC algorithm. The locality estimation mechanism is identical to the BCC CWS algorithm, except for the following changes. For each page an 'entry' bit is added to indicate if the page corresponds to a beginning of a new locality. On a page fault the entry bit of the demanded page is examined to see if the process is entering a new locality. If a new locality is detected the quantum is prematurely terminated. The next pre-load set is loaded according to the estimate for the new locality, as in the Bayesian algorithm.

The Predictive BCC algorithm was designed as a compromise between the BCC CWS and the Bayesian algorithms. The following points influenced our design:

- The Bayesian algorithm requires more information to be maintained, hence, the overhead, space as well as time, for implementation will be large as compared to the BCC CWS algorithm.
- Since we are exploring the possibility of implementation of algorithms using structural information on the BCC 500, such algorithms should require as few changes as possible.
- The BCC CWS algorithm makes use of recency of use information alone (and no a priori information) in selecting pre-load sets.
4.4.3 Simulation

We give below the results obtained in a simulation of the above mentioned pre-loading schemes. We simulated only the memory management part of the system. Once the process working set is loaded into the main memory, it is allowed to execute for a quantum, during which all its page faults are satisfied. In the case of the Bayesian and the Predictive BCC algorithms the quantum may be prematurely terminated if a change of locality is detected.

The data used for this simulation was an actual program trace of a compiler execution (provided by D.J. Hatfield of IBM); this trace consisted of a page reference string without "runs" of references to the same page. The parameters for the simulation are:

- number of instruction pages = 117
- number of data pages = 115
- window size = 40 (page references)
- quantum = 40 (page references)
- total references processed = 1800.

The simulation results are tabulated in Table 4.4.1 (where a number of parameters are also defined).

Cost Comparison

We define the cost of a pre-loading scheme as:

\[ C_L = \text{average time the process spends in loading state.} \]

\[ = \left( \frac{\text{average time for pre-loading}}{\text{the pre-load sets}} \right) + \left( \frac{\text{average time for loading}}{\text{pages on page faults}} \right) \]

\[ = n \cdot t_r \cdot n_{SI} + d(t_r / 2) \]  \hspace{1cm} (4.4.1)
Table 4.4.1
Simulation Results

<table>
<thead>
<tr>
<th></th>
<th>W.S. Algorithm</th>
<th>BCC CWS Algorithm</th>
<th>Bayesian Algorithm</th>
<th>Predictive BCC Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av. pre-load set size (p)</td>
<td>21</td>
<td>10</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>Av. swap-out set size (s)</td>
<td>21</td>
<td>26</td>
<td>20</td>
<td>23</td>
</tr>
<tr>
<td>Av. resident set size (w)</td>
<td>21</td>
<td>19</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>Total # of demanded pages i.e. page faults (d)</td>
<td>736</td>
<td>789</td>
<td>527</td>
<td>566</td>
</tr>
<tr>
<td>Total # of pages replaced (d_r)</td>
<td>725</td>
<td>66</td>
<td>367</td>
<td>59</td>
</tr>
<tr>
<td>Total # of swap-in operations (n_{sl})</td>
<td>48</td>
<td>48</td>
<td>68</td>
<td>67</td>
</tr>
<tr>
<td>Total # of swap-out operations (n_{so})</td>
<td>45</td>
<td>45</td>
<td>65</td>
<td>64</td>
</tr>
<tr>
<td>Total # of locality changes encountered</td>
<td>-</td>
<td>-</td>
<td>48</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 4.4.2
Costs & Drum Service Rates

<table>
<thead>
<tr>
<th></th>
<th>W.S. Algorithm</th>
<th>BCC CWS Algorithm</th>
<th>Bayesian Algorithm</th>
<th>Predictive BCC Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost (C_L)</td>
<td>416 t_r</td>
<td>447 t_r</td>
<td>331 t_r</td>
<td>350 t_r</td>
</tr>
<tr>
<td>Drum Service Rate</td>
<td>1.02 F</td>
<td>1.40 F</td>
<td>1.50 F</td>
<td>1.47 F</td>
</tr>
</tbody>
</table>
This is a reasonable criterion for cost because larger delays would necessitate a larger main memory for additional processes, in order to maintain the same degree of system performance. The time required to determine pre-load sets is neglected relative to $t_r$. The cost of swapping pages out may also be neglected, because (for BCC 500):

i) the number of pages which actually have to be written out are less than the average swap-out size, because only those pages which are modified need to be written back on the drum. In general about 30-50% of the data pages of the working set are modified.

ii) the drum transfer bandwidth (number of words transferred/second) is enough to take care of all swapping operations, i.e., for loading and unloading of processes, for loading demanded pages, and unloading pages replaced to make space for demanded pages.

We also evaluate the "throughput" possible, assuming the drum is the critical resource in the system, to see which algorithm puts the maximum load on the drum for swapping operations.

\[
\text{Drum Service Rate} = \frac{\text{average number of processes serviced per drum revolution}}{F} = \frac{S}{p+m+d'} F
\]

where

$F = \text{drum utilization} = \text{fraction of time a drum is busy reading or writing}$

$p = \text{average pre-load set size}$

$d' = \text{average number of pages demanded per schedule interval} = \frac{d}{n_{SI}}$
\[ m = \text{average number of modified pages written out} \]
\[ = 0.5 \left( s + \frac{d}{n_{SI}} \right) \cdot \frac{\text{number of data pages}}{\text{program size}} \]

(assume 50\% of data pages are modified)

\[ S = \text{total number of sectors on the drum} = 48 \]

We tabulate the cost \( C_L \) and the drum service rate for the above pre-loading algorithms in Table 4.4.2. (Note: In computation of cost \( C_L \), we have assumed \( n = 1; t_r \), for the BCC 500, is 34 msec.)

**Discussion**

From the above cost analysis, we notice that in the BCC CWS, Bayesian, and Predictive BCC algorithms, the drum service rate is almost the same, but the loading costs for the latter two algorithms are much lower. However, the number of pre-loading operations for these algorithms are much higher than those for the BCC CWS (or the Working Set) algorithm mainly because of the locality changes. It is conceivable that for some programs the number of pre-loading operations may become excessively high so that locality loading overheads may become intolerable. This situation may deteriorate to the equivalent of "thrashing" (in demand paging) where the system may just be doing loading and un-loading operations with little computation. To guard against such an occurrence, we recommend putting a constraint on the minimum amount of time allowed before loading a new locality. Consider the following two schemes.

**Scheme I:** Only test for locality changes at the end of a quantum.

This way no extra loading and unloading operations are needed. Also, the computational overheads for checking locality changes will be much
smaller since these need be performed only at the end of a quantum.

The performance of this scheme in general will not be worse than the BCC CWS algorithm because

i) the number of pre-loading operations is same in both cases

ii) some page faults may be saved due to pre-loading of the locality at the beginning of the quantum. However, the savings will not be appreciable because at the end of the quantum, most of the next locality may already have been loaded by demand.

This scheme would thus yield only marginal improvement in the cost, $C_L$, while guarding against the possibility of "thrashing."

**Scheme II:** Once a new locality is loaded, disallow swapping due to locality changes for at least 50-100% of its first quantum of execution. This will ensure that pre-loading activity is not excessive.

To evaluate this scheme, we perform the following analysis. Let

$$\begin{align*}
y &= \text{fraction of quantum during which locality swapping is disabled} \\
P_x &= \text{average number of pages of the pre-load set which are referenced before replacement} \\
\bar{E} &= \text{average locality execution duration} \\
T &= \text{quantum size} \\
\bar{n}_S &= \text{number of quantums (number of pre-loading operations) for the BCC CWS algorithm}
\end{align*}$$

Then we have the following cases.
Case I. T\cdot y > \bar{t}, i.e., localities change rapidly.

After every $T\cdot y$ time period a new locality will be loaded. The locality being pre-loaded will in general be in the midst of its execution.

Let $z = \text{fraction of execution period remaining for locality}
\approx \left\{\text{fraction of pages of the locality pre-load set yet to be referenced (assumed)}\right\}.$

Then $z \approx \frac{T\cdot y}{\bar{t}} - \left\lfloor \frac{T\cdot y}{\bar{t}} \right\rfloor.$

Also $n'_{SI} = \text{number of pre-loading operations for scheme II}$

$$n'_{SI} = \frac{n_{SI}}{y}$$

Hence extra cost of pre-loading = $\left\{\frac{n_{SI}}{y} - n_{SI} \right\} \cdot t_r$

and savings due to page fault cost = $(t_r/2) \cdot p_x \cdot z' \cdot (n_{SI}/y)$

$\Delta C_L = \text{added cost of scheme II over the BCC CWS algorithm}$

$$\Delta C_L = \frac{n_{SI}}{y} \cdot t_r \cdot \left\{(1-y) - p_x \cdot z/2\right\}$$

Thus if $p_x \cdot z \geq 2(1-y)$ scheme II will yield a lower cost than the BCC CWS algorithm, i.e., if the number of page faults saved per interval is greater than $2(1-y)$. If $p_x \cdot z > 2$, scheme II will always produce a lower cost $C_L$.

Case II. $T\cdot y < \bar{t} < T$, i.e., localities do not change rapidly.

After every $\bar{t}$ time period a new locality will be loaded. Here all page faults will be saved, due to pre-loading the locality at its beginning of the execution. Thus the added cost $\Delta C_L$ is given by

$$\Delta C_L = \frac{n_{SI}}{y} \cdot t_r \cdot \left\{(1-y) - p_x \cdot z/2\right\}$$
Due to the absence of factor z, the savings in page fault cost will be larger as compared to case I. Thus Scheme II will give a better performance for case II.

**Case III.** \( T < \bar{t} \), i.e., localities change slowly.

The savings in cost would be dependent on the number of locality changes that occur.

Let \( n_\lambda \) = number of locality changes.

Then savings in page fault cost due to pre-loading = \( \frac{t_r}{2} \cdot p_x \cdot n_\lambda \),

and the maximum extra cost of pre-loading operations = \( n_\lambda \cdot t_r \)

\[ \therefore \Delta C_L = n_\lambda \cdot t_r \cdot \{1 - p_x/2\} \]

Hence if \( p_x > 2 \), i.e., the number of page faults saved due to pre-loading is more than 2, Scheme II will perform better than the BCC CWS algorithm.

Thus for Scheme II the performance improves as \( \bar{t} \), the average execution period for localities, increases beyond \( T \cdot y \). Beyond this point, performance remains more or less steady with increases in \( \bar{t} \) until it drops for larger values \( \bar{t} \). Scheme II would generally be preferable to Scheme I, because in Scheme I there may be only marginal improvement in the cost relative to the BCC CWS algorithm. Simulation of Scheme II, for \( y = 0.85 \) and the data cited above yielded a cost of 367\( t_r \) (cf. Table 4.4.2).
4.4.4 Implementation Requirements

We discuss here space and computational requirements for implementation of the Bayesian and Predictive BCC algorithms in the BCC 500 computer system.

Memory Requirements for Bayesian. The information that needs to be maintained for the Bayesian algorithm is:

- the locality number information for 'entry' pages
- membership list for each locality
- mean and variance for each locality
- overall process variance
- current locality number
- the sample information, i.e.,
  - sum of distinct pages referenced in each quantum in current locality
  - number of such quantums in current locality

All this information needs to be maintained in the context block (CB) since all this is required while the process is active. Thus the space requirements considered below are with respect to data structures in the CB.

The locality number information needs to be kept in the PMT entry for 'entry' pages of each locality; since on a page fault this entry is looked up to locate the demanded page. We will first determine the maximum number of localities so that we can ascertain the number of bits required for storing the locality number information. For the BCC 500 the virtual space of each process is limited to 64 pages, however,
a process may have up to 90 physical pages in the system. Thus for a process size of 90 pages, and assuming a minimum locality size of 6 pages, we require 90/6 = 15 locality numbers. If a process has more than 15 localities then we must merge adjacent localities which jointly satisfy condition (2.4.1). If there still remain more than 15 localities then we consider only those which have higher estimates of execution times (cf. Section 2.4).

For a maximum of 15 localities, we can reserve 4 bits per PMT entry for the 'entry' pages of localities. Since we detect a locality change by checking if the demanded page is an 'entry' page, we require one bit per page to indicate whether the page is an entry page or not. The locality numbers themselves are kept in a separate table, parallel to PMT. Even though this results in a waste of CB space (since most of these entries will be unused), the locality numbers can be accessed rapidly. The space requirement for this is 128*4/24 words.

For locality membership information, we need to keep a list of pages belonging to each locality. Assuming an average of 10 pages per locality in the list, plus two entries per locality for the mean and variance, we have:

Total number of entries in locality list = (10+2)*15 = 180

Since we only need pointers into the PMT table, we require 8 bits per entry (see, e.g., the APT entry in Figure D-4).

The space requirement for sample and other information is as follows:

Process variance = 8 bits
Current locality number = 4 bits
Total distinct pages referenced = 12 bits
Number of time quantums = 8 bits

\[ \text{Total CB space required} = \frac{128 \times 4}{24} + \frac{180 \times 8}{24} + \frac{32}{24} \approx 82 \text{ words.} \]

Space used in the BCC 500 system for page-translation related data structures in the CB = 64 + 512 + 66 = 642 words.

\[ \text{Percent extra CB space required} = \frac{82}{642} = 12.7\%. \]

**Memory Requirements for Predictive BCC.** The information needed for Predictive BCC algorithm is as follows:

- the locality number information for 'entry' pages
- locality pre-load set size
- membership of pre-load sets
- current locality number

The space requirement in CB can be similarly estimated to be

\[ = \frac{128 \times 4}{24} + \frac{165 \times 8}{24} + \frac{4}{24} \approx 76 \text{ words.} \]

\[ \text{Percent extra space required (in the CB)} = \frac{76}{642} = 11.8\%. \]

**Computational Requirements for Bayesian.** There are two types of computational requirements we have to consider for predictive pre-loading algorithms, i.e.,

I. Computational requirement for obtaining a priori information

II. Computational requirements during execution, i.e., for checking locality changes, loading new localities, etc.

Preprocessing costs consist of costs incurred for collection of a priori information, i.e., for determination of localities, their sizes, etc. If we assume that no reordering (or code optimization) of program
code and data is carried out, then the cost of this step is appreciable. The cost in this case will be proportional to the cost of locating SCRs, which according to Aho, et. al. [44], is proportional to $\text{Max}(n,e)$, for an $n$ node and $e$ edge program graph. However, if we assume that reordering (or code optimization) is performed the (additional) preprocessing costs would be minimal, since this a priori information is generated and used for reordering (and code optimization) itself.

The execution time costs are more important and need to be considered in detail. The various computational requirements during execution of a program consist of the following:

- **Checking locality changes:** A change of locality is sensed by checking the entry bit of each new demanded page, and if 'true' then comparing its locality number with the current locality number. This checking can be done, upon page faults, while the locations of new pages are being determined.

- **Loading a new locality:** If a new locality is sensed, the quantum of the process is terminated. The modified pages of the resident set are unloaded to the auxiliary memory and the CWS bit of APT entries corresponding to pages of the new locality are set. Thus when the process is next loaded the pages of the new locality are loaded as the pre-load set. The extra overhead consists of obtaining the pages of the new locality from the membership list and setting the CWS bits in corresponding APT entries.

- **Updating sample information:** At the end of each quantum
the total number of distinct pages referenced in the quantum are added to the sum of distinct pages referenced in the current locality (i.e., to Sample_Total). This information is obtained by checking the RF (reference flag) bit in the PMT entry and resetting it for the next quantum. The number of quanta (N) in the current locality is updated by adding one. At the beginning of a new locality, the values are initialized as follows:

\[ N = n' = \frac{\text{Process Variance}}{\text{Locality Variance}} = \frac{\sigma^2}{\sigma_j^2} \]

\[ \text{Sample_Total} = m_j' * n' \]

where \( m_j' \) and \( \sigma_j' \) are the a priori mean size and variance estimates for the \( j^{th} \) locality.

- Locality estimation: The locality size (LS) at the end of a quantum is estimated as follows:

\[ \text{LS} = \frac{\text{Sample_Total}}{N} \]

The membership of the locality is determined using the use history (UH) bits of pages, by loading the most recently referenced pages (or as per the replacement algorithm).

**Computational Requirements for Predictive BCC.** The Predictive BCC algorithm uses most of the ideas of the BCC CWS algorithm in addition to the locality pre-loading concept of the Bayesian scheme, as explained earlier. The execution time overheads are limited to the following operations:

- Checking locality changes: Checking the 'entry' bit of a
demanded page (upon page faults) to see if a new locality needs to be loaded, similar to that for Bayesian.

- Loading a new locality: If a new locality is sensed then pages corresponding to the new locality constitute the pre-load set for the next quantum. The operations required are identical to those explained for the Bayesian.

4.4.5 Feasibility of Predictive BCC

Here we consider the feasibility of implementation of the Predictive BCC algorithm on the BCC 500 computer system. For this particular system, the Bayesian algorithm appears infeasible for the following reasons:

- the overhead for both space and computation is higher
- implementation would require a larger number of changes to the existing system, necessitated due to the nature and working of the Bayesian algorithm, as compared to the BCC CWS algorithm.

The Predictive BCC algorithm, on the other hand, uses all the data structures of the BCC CWS algorithm with the addition of locality information. Thus it requires a minimum amount of change in the system structure for implementation. The extra space required as well as the computational overheads can be satisfied, as discussed below.

The space overheads for the Predictive BCC algorithm consist of 55 words for locality membership lists, 1 bit per page for 'entry' bit, 4 bits per 'entry' page for locality number. The space for membership lists can be satisfied by allocating 55 words of the FREE space between
address 10 - 177 (octal) in the CB. (See Figure D-5.) The one 'entry' bit per page required for checking locality change, can be provided as one of the unused bits in words 3 or 4 of the PMT entry for each page. For keeping the locality number information there are two alternatives. The first alternative is to keep a table parallel to PMT (as suggested earlier) with 4 bits per page. In this table only the entries corresponding to 'entry' pages will be used. The advantage of this scheme is fast access to the locality number. The other alternative is to keep a list with PMT index and locality number for entry pages alone. This will result in saving of an appreciable amount of space (since the number of entry pages is usually small), but would increase the access time since the list may have to be searched sequentially in order to find the locality number for an entry page. Both of these alternatives are feasible and the extra space requirement can also be accommodated in the FREE space mentioned above.

For computational overheads let us consider the memory management of the BCC 500. The memory management tasks are delegated to a separate processor, the AMC, which takes care of the swapping operations, the page faults, the allocation of memory resources, etc. The AMC is best suited for implementation of the changes required for the Predictive algorithm since the AMC is microprogrammed and it is relatively simple to implement changes in memory management algorithms. The computational load on the AMC would increase. However, the overhead due to extra computations required at page faults will be minimal because the total number of page faults will be substantially reduced (for the example considered, the reduction is over 20% for Scheme II, with
y = 0.85. In addition, the number of extra operations required at each page fault is small (one indexing operation, if there is no locality change). However, in the event of a locality change the computational overhead increases, since the CWS bits for pages corresponding to the new locality need to be set (in APT entries). This operation is also carried out for the BCC CWS algorithm, only there the pre-load set is determined in a different manner (using the UH bits for active pages). In the Predictive BCC algorithm the CWS bits are set using the membership list for pages belonging to the new locality. The computational overheads for this operation are greater than those for the BCC CWS algorithm, but should still be much less than \( t_r \) (cf. Eq. (4.4.1)).

The main difference in \( C_L \) will be in the number of times this operation would be performed (\( n_{SI} \)). For the Predictive algorithm, the maximum number of such operations may be \( 1/y \) times the corresponding operations in the BCC CWS algorithm, where \( y \) is the fraction of quantum during which locality change is disabled. (Simulation of Scheme II for \( y=0.85 \) increased \( n_{SI} \) by about 15%, but decreased \( C_L \) by about 18%.) The extra computational requirement can be handled by the AMC since it can execute up to \( 5 \times 10^6 \) microinstructions per second and currently the load on the microprocessor is much lower than that.

Thus both computational and space overheads can be satisfied by the BCC 500 system making the implementation of the Predictive BCC algorithm feasible.
An Example Of Matrix Calculations In A Paging System

We consider here an example of considerations involved in performing matrix calculations in a paging environment. The arrangement of elements of a matrix is a paramount problem. If the matrix is arranged in a row order, the elements may be referenced in column order, or vice versa, producing a large number of page faults. One should therefore store matrices according to their use if possible. Alternatively, or when matrices are used in several ways, one should design algorithms to take into account the storage mode.

Consider the problem of multiplication of two matrices. Suppose matrices Z(N,N), X(N,N), and Y(N,N) are stored in a row order, and that the page size is N, so that there are N pages per matrix, each page containing one row. To calculate \( Z = X \cdot Y \), we have (canonically)

\[
\begin{align*}
& \text{DO } I=1 \text{ TO } N; \\
& \hspace{1em} \text{DO } J=1 \text{ TO } N; \\
& \hspace{2em} Z(I,J) = 0; \\
& \hspace{2em} \text{DO } K=1 \text{ TO } N; \\
& \hspace{3em} Z(I,J) = Z(I,J) + X(I,K) \cdot Y(K,J); \\
& \hspace{3em} \text{END;} \\
& \text{END;} \\
& \text{END;}
\end{align*}
\]

If we assume one page per matrix is allowed to remain in the main memory, the total number of page faults produced (due to references to matrices X, Y, and Z) by this algorithm is given by:

\[
N \text{ (for } X) + N \cdot N^2 \text{ (for } Y) + N \text{ (for } Z) = N^3 + 2N.
\]

The number of page faults can be substantially reduced if the algorithm is varied slightly to make references to elements of each matrix in the
order they are stored, as much as possible.

Consider the following modification to the matrix multiply algo-

**rithm:**

```plaintext
DO I=1 TO N;
    DO J=1 TO N;
        Z(I,J) = 0;
    END;
    DO J=1 TO N;
        DO K=1 TO N;
            Z(I,K) = Z(I,K) + X(I,J) * Y(J,K);
        END;
    END;
END;
```

The algorithm multiplies the first element of X with the first row of Y and stores partial results in the first row of Z and so on. The number of operations and address calculations remain the same, while the number of page faults (for matrix pages alone) are reduced to:

\[ N \text{ (for } X) + N \times N \text{ (for } Y) + N \text{ (for } Z) = N^2 + 2N. \]

Thus by varying the algorithm slightly, the page residency may be increased and hence the total number of page faults reduced.
APPENDIX B

Counter Example To Show That $C_D$ (In Section 2.3) Is Not Monotonic

Consider the following example:

Let $z =$ page size in number of data nodes, $z > 1$.

$r =$ number of localities, $r > 1$.

$m =$ total number of data nodes considered for pagination, so far

$k = \left\lfloor \frac{m}{z} \right\rfloor =$ total number of data pages required for $m$ data nodes

$c(m) =$ cost $C_D$ for $m$ nodes

\[
\begin{align*}
&= t_w \left\{ t_{\ell_i} \sum_{i=1}^{r} \tau_{ij} p_i(m) \{ t_{\ell_i} w + q_{ji}(m) \} \right\} \quad (B.1) \\
\end{align*}
\]

where

\[
\begin{align*}
p_i(m) &= \lvert Q^i_i \rvert \text{ for m data nodes considered.} \\
q_{ji}(m) &= \lvert Q^i_j \wedge \overline{Q}^i_i \rvert \text{ for m data nodes considered.} \\
\end{align*}
\]

Assume that up to the assignment of $m - 1$ data nodes, the $k^{th}$ page contains data nodes referenced by all localities, except $\ell_u$. Also assume that the $m^{th}$ data node is referenced by the $\ell_u$ alone. Then on assignment of the $m^{th}$ data node to the $k^{th}$ data page, we obtain

\[
\begin{align*}
p_i(m) &= p_i(m-1), \text{ for all } i \neq u, \\
p_u(m) &= p_u(m-1) + 1, \quad (B.2) \\
q_{jj}(m) &= q_{jj}(m-1) = 0, \text{ for all } j, \\
q_{ju}(m) &= q_{ju}(m-1) - 1, \text{ for all } j \neq u, \quad (B.3) \\
q_{ji}(m) &= q_{ji}(m-1) \quad \text{, otherwise.} \\
\end{align*}
\]

Then
\[ c(m) = t_w \left\{ \sum_{i \neq u} \sum_j r_{ij} p_i(m) \left( \frac{t_{u_j}}{t_w} + q_{ji}(m) \right) + \sum_j r_{uj} p_u(m) \left( \frac{t_{u_u}}{t_w} + q_{ju}(m) \right) \right\} \]

\[ = t_w \left\{ \sum_{i \neq u} \sum_j r_{ij} p_i(m-1) \left( \frac{t_{u_j}}{t_w} + q_{ji}(m-1) \right) + \tau_{uu} \left( p_u(m-1) + 1 \right) \left( \frac{t_{u_u}}{t_w} + q_{ju}(m-1) - 1 \right) \right\} \]

\[ = t_w \left\{ \sum_{i \neq u} \sum_j r_{ij} p_i(m-1) \left( \frac{t_{u_j}}{t_w} + q_{ji}(m-1) \right) + \tau_{uu} \frac{t_{u_u}}{t_w} + \sum_j r_{uj} \left( \frac{t_{u_j}}{t_w} + q_{ju}(m-1) - p_u(m-1) - 1 \right) \right\} \]

or \[ c(m) = c(m-1) + t_w \left\{ \sum_j r_{uj} \frac{t_{u_j}}{t_w} + \sum_{j \neq u} r_{uj} q_{ju}(m-1) \right\} \]

Thus if \( \left\{ \sum_j r_{uj} \frac{t_{u_j}}{t_w} + \sum_{j \neq u} r_{uj} q_{ju}(m-1) \right\} < (p_u(m-1) + 1) \), then \[ c(m) < c(m-1) \]. One case in which this holds is when locality \( l_u \) references all of the k data pages and \( t_{l_u} >> t_w \). Hence the cost function \( C \) is not necessarily a monotonically nondecreasing function of \( m \), the number of nodes considered.
APPENDIX C

Generalization Of (3.1.13) For Arbitrary N

For \( N = 4 \), \( r_1 \neq x_1 \), and \( h(k) = 1, \forall k \), the expression for cost functional \( f \) is given by

\[
f_4(x_1, r_1, 1) = 1 + \min_{y_1} \left[ \sum_{r_2 \in x_2} \left( \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_2, r_4, 4) \right) \right) \right) \right] + \sum_{r_2 \in x_2} \left( 1 + \min_{y_2} \left[ \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_3, r_4, 4) \right) \right] \right) + \sum_{r_2 \in x_2} \left( 1 + \min_{y_2} \left[ \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_4, r_4, 4) \right) \right] \right) (C.1)
\]

Then the optimal replacement decision \( y_1^* \) is given by (using the end condition (3.1.7)):

\[
y_1^* = \text{ARG} \left\{ \min_{y_1} \left[ \sum_{r_2 \in x_2} \left( \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_2, r_4, 4) \right) \right) \right) \right] + \sum_{r_2 \in x_2} \left( 1 + \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_3, r_4, 4) \right) \right) + \sum_{r_2 \in x_2} \left( 1 + \sum_{r_3 \in x_2} \left( \sum_{r_4 \in x_2} f_4(x_4, r_4, 4) \right) \right) \right] \right) (C.2)
\]

where

\[
x_2 = x_1 - y_1 + r_1
\]
\[
x_3 = x_2 - y_2^* + r_2
\]
\[
x_4 = x_3 - y_3^* + r_3
\]

and
We can rewrite expression (C.2) as

\[
y_2^* = \text{ARG} \left\{ y_2 \min \left\{ \sum r_2^2 y_2 + \sum r_3^2 y_2 + \sum r_3^2 y_2 + \sum r_3^2 y_2 + \sum r_3^2 y_2 \right\} \right\}
\]

\[
y_3^* = \text{ARG} \left\{ y_3 \epsilon x_3 \min \left\{ r_3^2 y_3 \right\} \right\}
\]

\[
y_3^{*} = \text{ARG} \left\{ y_3^{*} \epsilon x_3 \min \left\{ r_3^2 y_3^{*} \right\} \right\}
\]

\[
x_3^{*} = x_2 + r_2 - y_3^*(y_2)
\] (C.4)

Using relations (C.3) to expand the third, and the last three terms, we have

\[
y_1^{*} = \text{ARG} \left\{ y_1 \min \left\{ \sum r_2^2 r_2 \sum r_3^2 r_3 \sum r_4^2 r_4 + \sum r_4^2 r_4 + \sum r_4^2 r_4 + \sum r_4^2 r_4 + \right\} \right\}
\]
Combining the first, third, sixth, and tenth terms together and second and eighth terms together, we obtain

\[ y^*_1 = \text{ARG} \left\{ \min \left\{ \sum_{r_2} p_{r_1 r_2} r_2 \sum_{r_3} p_{r_2 r_3} r_3 \sum_{r_4} p_{r_3 r_4} + \sum_{r_2} p_{r_1 r_2} \sum_{r_3} p_{r_2 r_3} + \right. \right. \]

\[ \left. \left. \sum_{r_2} p_{r_1 r_2} \sum_{r_3} p_{r_2 r_3} \right\} \right\} \]

Combining the sixth and ninth terms together, we obtain

\[ y^*_1 = \text{ARG} \left\{ \min \left\{ \sum_{r_2} p_{r_1 r_2} r_2 \sum_{r_3} p_{r_2 r_3} r_3 \sum_{r_4} p_{r_3 r_4} + \sum_{r_2} p_{r_1 r_2} \sum_{r_3} p_{r_2 r_3} + \right. \right. \]

\[ \left. \left. \sum_{r_2} p_{r_1 r_2} \sum_{r_3} p_{r_2 r_3} \right\} \right\} \]
Considering terms dependent upon $y_1$ alone, we have

$$y_1^* = \arg\left(\min\left\{ \sum_{r_1} p_{r_1} r_1 \sum_{r_2} p_{r_2} r_2 \sum_{r_3} p_{r_3} r_3 \right\} \right)$$

Combining the second, tenth, eleventh, and seventeenth terms together, the third, seventh, and eighteenth terms together, the fifth, fourteenth, and fifteenth terms together, and the ninth, twelfth, thirteenth, and nineteenth terms together, we obtain (using $x_2^2 = x_1^2 + r_1 - y_1^*$)

$$y_1^* = \arg\left(\min\left\{ \sum_{r_1} p_{r_1} r_1 \sum_{r_2} p_{r_2} r_2 \sum_{r_3} p_{r_3} r_3 \right\} \right)$$
\[ p_{r_1 y_1} \left( \sum p_{r_2 r_3} r_4 \sum p_{r_3 r_4} r_3 \sum p_{y_1 r_3} p_{r_3 r_3} \right) + \]
\[ \sum p_{r_1 r_2} p_{r_2 y_1} (p_{r_2 r_2} - p_{y_1 y_1}) + \sum p_{r_1 r_2} p_{r_2 y_1} + \]
\[ p_{r_1 y_1} \sum p_{y_1 r_3} + \sum p_{r_1 r_2} \sum p_{r_2 r_3} p_{r_3 y_3} + p_{r_1 y_1} + \]
\[ \sum p_{r_1 r_2} \sum p_{r_2 r_3} p_{r_3 y_3} \left( \sum p_{r_1 r_2} p_{r_2 y_1} + p_{r_2 y_2} + \right) \]
\[ \sum p_{r_1 r_2} \sum p_{r_2 r_3} p_{r_3 y_3} \right) \]  

(C.6)

If, as in Section 3.1, we assume that programs satisfy the locality property (3.1.11) and the programs are managed accordingly, (and that the \( p_{y_1 y_1} \), \( y_1 y_1 \), are large and of similar magnitude), then the second, third, fifth, sixth, eighth, and the ninth terms may be neglected. Hence (C.6) may be rewritten as

\[ y_1^* = \text{ARG} \left\{ p_{r_1 y_1} + \sum p_{r_1 r_2} p_{r_2 y_1} + \right\} \]
\[ \sum p_{r_1 r_2} \sum p_{r_2 r_3} p_{y_3 y_3} \]  

i.e. the optimal page to be replaced, at the current step, is one which minimizes the probability of reference, from the currently referenced page, along all paths of length \( < N (\approx 4) \).

Extending the above, this can be shown to be true for an arbitrary \( N \), i.e., in general

\[ y_1^* = \text{ARG} \left\{ p_{r_1 y_1} + \sum p_{r_1 r_2} p_{r_2 y_1} + \right\} \]
\[ \sum_{r_1^2 y_1} p_{r_1 r_2} \sum_{r_2^2 r_2} p_{r_2 r_3} p_{r_3 y_1} + \cdots + \]
\[ \sum_{r_1^2 y_1} p_{r_1 r_2} \sum_{r_2^2 r_2} p_{r_2 r_3} \cdots \sum_{r_{N-1}^2 r_{N-2}} p_{r_{N-2} r_{N-1}} p_{r_{N-1} y_1} \} \] (C.7)
APPENDIX D

BCC 500 System

The BCC 500 system is oriented towards giving fast response to a large number of interactive users. For this purpose its organization and operation is different from more conventional systems. The set of algorithms and data structures which allow the BCC 500 to run as a time-sharing system are distributed over a number of independent processors and pieces of software. (they are collectively known as the BCC 500 operating system.) The independent processors enable a number of activities to be carried out simultaneously so that their efficient use leads to high throughput for the system. In the following description we will limit ourselves to components of the system which are involved in memory management. The main components of the system are:

- CPU--central processing unit used mainly for computing (2 units)
- µ-Scheduler--microprocessor for scheduling which decides what processes to schedule and whether to preempt a running process etc.
- AMC--microprocessor for memory management
- Core--256K words (24 bit words) of interleaved core memory with 4 access ports
- Drums & Disks--auxiliary memory; drums serve as the secondary memory while disks act as backup and storage for large files, etc.

The descriptions that follow are taken from [2, 3, 60].

MEMORY MANAGEMENT OF BCC500

The AMC (Auxiliary Memory Control) microprocessor is mainly responsible for memory management. This microprocessor executes a 90 bit
instruction with average instruction execution time of approximately 200 nsec. The main advantage of using a separate microprocessor are:

- the microprocessor code can be adapted to memory management needs alone.
- the speed of execution is high.
- the CPU is relieved of the load of memory management.

The AMC communicates with the drums and disks through a controller, the AMTU (Auxiliary Memory Transfer Unit), to which the drums and disks are attached. The controller keeps track of the positions of the read heads on the drums and disks at all times to facilitate information transfer. Data is recorded on drums and disks in the units of 2K word pages, prefixed by a 48 bit unique name (UN). Each page in the system has a UN which identifies the user name, the file name within user files, the page number etc.

Each drum has a very high transfer rate (710 pages/sec.), to keep the pages flowing through main memory rapidly. For executing a large number of processes simultaneously, the processes must be brought in at a fast rate. The drum is allocated dynamically, so that a process core working set can be transferred onto a drum in minimum possible time. We consider this in more detail in Section 4.3. As processes become active they are moved from drum to core, and back out on drum at the end of the quantum.

Particular drum specifications include

- 24 sectors around the drum
- 40 bands across the drum, therefore 24×40=960 pages per drum
- Drum transmits one word in parallel
34 msec. rotation time
550 nsec. to transfer one word
30 µsec. inter-sector gap.

For each process only its current CWS (Core Working Set) is kept in main memory. The remaining pages are either kept on the drum or on the disk, depending upon the page activity. A referenced page may be anywhere in the system, hence some form of mapping is required to locate a referenced page.

The mapping of virtual addresses to physical addresses goes through a number of steps, depending upon where the actual page is located. The various data structures used for mapping are:

**PM--Physical Map:** 128 hardware registers (since a process can reference a maximum of 128 pages at any time) which contain the physical address of the corresponding virtual page, if it is in core memory and has been referenced at least once.

**CHT--Core Hash Table:** a table of all pages in core memory; hashed on the unique name of the page.

**DHT--Drum Hash Table:** a table of all pages on drum; hashed on the disk address of the page (the disk address for a page is constant, while its drum address varies).

**PMT--Process Memory Table:** contains the unique name, the disk address, and related data for each known page of a process.

**MAP--Process Map:** 128 half word entries, each entry giving the PMT index of the corresponding virtual page.
MAPPPING

On a memory reference the higher 7 bits of the virtual address are used as the virtual page number and are used to index into the PM to find the physical address of the page. If the corresponding entry in PM is non-zero, it gives the physical address. This condition is depicted in Figure D-1. If the entry in PM is zero then there are two possibilities;

Page in core but not in PM. Then the CHT has to be searched to find the physical location of the page. Two levels of translation are required in locating the page, i.e.

- from virtual page number to location independent name
  (the unique name (UN)).
- from UN to the physical address.

This is shown in Figure D-2. The UN of the page is used to hash into the CHT. If the page is in core, the PM entry is also set so that further references need not go through this translation mechanism. Thus even though the CWS is pre-loaded into the core, the physical map is loaded only "on demand."

Page not in core. The PMT entry for the virtual page is used to hash into the DHT which contains the drum address of the page, and the AMC is notified to transfer the page. If the page is not on drum, it has to be fetched from the disk unit.

The CHT and DHT are resident in core permanently because all the processes use them very frequently. The PMT and the Map are resident (in the "context block" for the process) while the process is in the core.
Figure D-1 - Mapping via PM

Figure D-2 - Mapping via PMT

UN

Disk Address (DKA)

A PMT Entry

RO: Read Only
SF: Sched. Flag
RF: Ref. Flag

A MAP Entry

Figure D-3

Use
History
Page
Lock
PMT
Index

DWS
LOCK
CWS
KEEP

An APT Entry

Figure D-4
CB CONTENTS

0: POP entry indirect address word
1: POP entry indirect address word
2: SP first unused stack address
3: SL last word allocated for stack
4: P for Trap (ring dependent)
5: PAR for Trap (ring dependent)
6: BRU for Trap (ring dependent)
7: BRU for Trap (ring dependent)

FREE: 10-177  (=119 words)

*MAP: 200-277 - Map associated with virtual pages
       (=64 words)

*PMT: 300-1277- Table of pages belonging to this process
       (=512 words)

SPT: 1300-2117
SPCS: 2120-2237
ICT: 2240-2307
OFT: 2310-2371
STACK: 2372-2647

*APT: 2650-2751 - Table of active pages
       (=66 words)

TRSTATE: 2752-2763
AWSTATE: 2764-2775
CTC: 2776
IT: 2777

Figure D-5
The context block contains most of the page mapping information for the process, as well as other tables and data structures used by the operating system. Every process created has a CB associated with it. The CB contains the "state" of the process at any time, and is resident in core when the process is executing. The contents of the CB are shown in Figure D-5. Details of an entry in PMT and MAP are shown in Figure D-3.

The AMC takes care of most of the memory management tasks, leaving the CPU to concentrate on computing. High "throughput," the rate at which user program requests are satisfied, is obtained by keeping the CPU reasonably busy, i.e. by feeding the CPU with enough processes to work on. This way whenever a process blocks, another process is available for immediate execution. Since processes may block frequently, a large number of processes are needed in core within short times. This is achieved by supporting a large amount of swapping activity through a fast drum and the AMC. In this respect BCC differs from conventional systems where swapping activity is kept low since the CPU also takes care of some of the swapping activity. As a result the process working sets tend to remain in memory for longer periods of time. Thus for comparable response a larger amount of main memory is required so that a larger number of processes can be kept in main memory. The main memory utilization for the BCC 500 system is better because the residence time of process (core) working sets is close to the quantum, since the swapping activity puts no constraint on the CPU.
CORE WORKING SET

The core working set (CWS) strategy refers to loading and maintaining in core a set of pages which have a high probability of executing a guaranteed quantum, while the process is in core. The information used for constructing CWS is the recency of use. The CWS, $W_s(t_i)$, at time $t_i$ consists of pages which were recently referenced by the process, and is defined as:

$$W_s(t_i) = \begin{cases} W_s(t_{i-1}) + x(t_i), & x(t_i) < W_s(t_{i-1}) \\ W_s(t_{i-1}) + x(t_i) - y(t_i), & \text{otherwise} \end{cases}$$

where $x(t_i) = \text{page currently referenced, i.e. at } t_i$, and $y(t_i) = \text{page with minimum value of use history at } t_i$, and

$\text{Max CWSS} = \text{maximum CWS size allowed for the program.}$

The use history (UH) is a representation of the recency of use information and it reveals the intervals, among the last 8 execution intervals (an execution interval here is defined as an interval of continuous execution), during which the corresponding page was referenced. UH is 8 bits updated after every execution interval of the process, by shifting the bits rightward by one position and setting the leading bit to '1' if page was referenced during the previous interval. RF flag in the PMT entry is used for this purpose.

Every time a process page faults the $\mu$-Scheduler is given control, which then decides whether the demanded page should be loaded or not. Also, at the end of a quantum the $\mu$-Scheduler decides whether to swap

---

1Note that the CWS scheme currently implemented on the BCC 500 is a very simplified version of the one described here.
the process out or to give it another quantum. If the process is dis-
missed, the pages occupied by the process are released to the system
and the pages which were modified during execution are written back on
the drum. Also CWS bit in the APT entries corresponding to the CWS
pages to be pre-loaded at the beginning of next quantum are set. The
CWS to be pre-loaded at next quantum consists of pages which were
referenced at least once in the last 'N' (N < 8) execution intervals.
The CWS information is kept in the active page table (APT) in the
context block. An APT entry is shown in Figure D-4.
ORDER: PROCEDURE OPTIONS

/* THE INPUT TO THE ALGORITHM IS MODIFIED ADJACENCY REACHABILITY
   MATRICES, THE PRESENT GRAPH, AND THE SCR TO NODE MAPPING FOR
   VARIOUS SCR'S. THE ADJACENCY(REACHABILITY) MATRIX IS MODIFIED
   IN THE SENSE THAT NON-ZERO ELEMENTS CORR. TO ARCS WHICH NEED TO
   BE REMOVED(FOR SCP'S), FOR ORDERING PURPOSES, ARE MADE EQUAL TO
   ZERO. ALSO THE END NODES OF THE REMOVED ARC ARE DESIGNATED AS
   ENTRY & EXIT NODES OF THE SCR( FOR USE IN ENTRY(EXIT LISTS).
   ALL NODES POINTING INTO(OUT OF) THE SCR POINT INTO(OUT OF) THE
   ENTRY(EXIT) NODES SO DESIGNATED. THE PRESENT GRAPH IS A BOOLEAN
   VECTOR WITH 1'S IN POSITIONS CORRESPONDING TO NODES OR SCR'S
   PRESENT IN THE REDUCED GRAPH. */

DECLARE ( N, /* TOTAL NO. OF NODES IN THE GRAPH */
      R, /* TOTAL NO. OF SCR'S IN GRAPH */
      NODE, /* CURRENT NODE BEING ORDERED */
      ORD_PNTR, /* NO. OF NODES IN ORDERED LIST */
      LEVEL, /* NESTING LEVEL AT WHICH CURRENT */
              /* ORDERING IS BEING DONE */
      I, J, K, L, M, N, /* TEMPORARIES */
      ) FIXED BINARY(31,0);

DECLARE ( ORDEREDIN, /* LIST OF ORDERED NODES */
       ADJNCY(N,N), /* ADJACENCY MATRIX FOR THE GRAPH */
       RECHBLTY(N,N), /* REACHABILITY MATRIX FOR THE GRAPH */
       OPEN(N), /* PRES_GRAPH NODES YET TO BE ORDERED */
       OPN_PNTR, /* POINTER INTO THE OPEN LIST */
       PRESENT_GRAPH(N), /* LIST OF NODES UNDER CONSIDERATION */
       ENTRY_LIST(R), /* ENTRY NODES FOR SCR'S */
       EXIT_LIST(R), /* EXIT NODES FOR SCR'S */
       ENTRY_NODE,EXIT_NODE ,NEXT_NODE ,
       SCR_ND_MAP(R,N), /* SCR TO NODE MAP */
       ) FIXED BINARY(31,0) CONTROLLED;

ALLOCATE ENTRY_NODE,EXIT_NODE,NEXT_NODE,OPN_PNTR ;
GFT LIST (N,R,ENTRY_NODE,EXIT_NODE) ;
N1 = N + R ;
ALLOCATE OPEN,PRESENT_GRAPH,ADJNCY,RECHBLTY,SCR_ND_MAP ;
ALLOCATE ORDERED,ENTRY_LIST,EXIT_LIST ;
GET SKIP EDIT(ADJNCY,RECHBLTY)
   ((N) (COL(1), (N) F(1)), (N) (COL(1), (N) F(1))) ;
GET SKIP EDIT(PRESENT_GRAPH,SCR_ND_MAP)
   ((N1) F(1), (R) (COL(1), (N1) F(1))) ;
GET SKIP EDIT(ENTRY_LIST,EXIT_LIST)((R) F(4), COL(1), (R) F(4)) ;

NEXT_NODE = 0 ;
OPN_PNTR = 0 ;
ORD_PNTR = 0 ;
OPFN = 0 ;
NODE = ENTRY_NODE;
PRESENT_GRAPH(NODE) = 0;
ORDFRDO = 0;
CALL ND_ORDER;
PUT SKIP EDIT(' ORDERED LIST OF INSTRUCTION NODES ..... ')((A));
PUT SKIP EDIT(ORDERED) .((N) F(4));

ND_ORDER: PROCEDURE RECURSIVE;
DO WHILE (I) :
  /* CHECK THE OPEN LIST TO SEE IF THERE IS A NODE FROM WHICH CUR*/
  /* NODE IS REACHABLE. IF SO ORDER THAT NODE PRIOR TO ORDERING */
  /* THE CURRENT NODE, O.K. ORDER THE CURRENT NODE. ----------- */
  L = 1;
  DO WHILE (L=1);
    L = O;
    K = NODE;
    DO WHILE (K>N);
      K = ENTRY_LIST(K-N);
    END;
    I = OPN_PNTR;
    DO WHILE(I>OGN=O);
      M = OPEN(I);
      DO WHILE (M>N);
        M = EXIT_LIST(M-N);
      END;
      IF RECHLTY(M,K)=1 THEN DO;
        /* A NODE IN OPEN LIST IS FOUND FROM WHICH CURRENT */
        /* NODE IS REACHABLE, HENCE ORDER THAT NODE FIRST, AND*/
        /* PUT THE CURRENT NOE ON OPEN LIST. ----------- */
        OPN_PNTR = OPN_PNTR + 1;
        OPEN(OPN_PNTR) = NODE;
        NODE = OPEN(I);
        /* COMPACT THE OPEN LIST ----------- */
        OPN_PNTR = OPN_PNTR - 1;
        DO J = I TO OPN_PNTR;
          OPEN(J) = OPEN(J+1);
        END;
        OPEN(OPN_PNTR+1) = 0;
        L = 1;
      END;
      I = I - 1;
    END;
  END;
  /* THERE IS NO NODE ON OPEN LIST FROM WHICH CURRENT NODE IS */
  /* REACHABLE. HENCE ORDER THE CURRENT NODE & SELECT AN ADJACENT*/
  /* NODE TO BE CONSIDERED FOR ORDERING NEXT. --------- */
  IF NODE -> N THEY DO ;
/* NODE IS NOT AN SCR, HENCE ORDER IT & PROCEED */

ORD_PTR = ORD_PTR + 1;
ORDERED(ORD_PTR) = NODE;

if NODE=EXIT_NODE then do:
    /* ALL NODES OF PRESENT_GRAPH HAVE BEEN ORDERED, GO BACK */
    /* ONE LEVEL & ORDER GRAPH AT THAT LEVEL */
    FREE OPEN, PRESENT_GRAPH, ENTRY_NODE, EXIT_NODE, ORD_PTR;
    FREE NEXT_NODE;
    Return;

end;

/* SELECT AN ADJACENT NODE TO BE ORDERED. */

NODE = NEXT_NODE;
end;

else do:
/* NEXT NODE TO BE ORDERED IS AN SCR -- REMOVE THIS FROM LIST */
/* INCREASE THE LEVEL BY ONE & SET THE ENTRY & EXIT NODES */
/* FOR THE NEW PRESENT_GRAPH WHICH CORRESPONDS TO THE SCR, */
/* AND APPLY THE PROCEDURE RECURSIVELY. */
/* SELECT THE NODE TO BE ORDERED (NEXT_NODE) AFTER ALL NODES*/
/* OF SCR HAVE BEEN ORDERED. */

J = NODE;
NODE = EXIT_LIST(J-N) ;
NEXT_NODE = NXT_NODE ;
NODE = J;
allocate OPEN, PRESENT_GRAPH, ENTRY_NODE, EXIT_NODE, ORD_PTR ;
allocate NEXT_NODE ;
ORD_PTR = 0;
OPEN = 0;
PRESENT_GRAPH = SCR_NODE_MAP(NODE-N); ENTRY_NODE = ENTRY_LIST(J-N) ;
EXIT_NODE = EXIT_LIST(J-N) ;
NODE = ENTRY_NODE ;
PRESENT_GRAPH(NODE) = 0 ;
call NO_ORDER ;
NODE = NEXT_NODE ;
end;

end ; /* OF DO WHILE(1) */
return ;

NXT_NODE: PROCEDURE RETURNS (FIXED BINARY(31,0)) ;
/* SELECTS A NODE TO BE ORDERED NEXT BY CONSIDERING THE */
/* PRESENT GRAPH & THE ADJACENCY MATRIX */

K = 0 ;
do M = 1 to N1 ;
    if PRESENT_GRAPH(M) > 0 then do ;
        I = M ;
        do while (I > N) ;
            I = ENTRY_LIST(I-N) ;
        end ;
    end ;
/* CHECK FOR ADJACENCY -------------------------------------------- */

IF ADJACENCY(INODE, I) = 1 THEN DO;
    K = K + 1;
    IF K = 1 THEN L = M; ELSE DO;
    /* PUT REMAINING ADJACENT NODES ON OPEN LIST -- */
    OPN_PTR = OPN_PTR + 1;
    OPEN(OPN_PTR) = M;
    END;
    PRESENT_GRAPH(M) = 0;
END;
RETURN (L);
END NXT_PTR;
END ND_ORDER;
END ORDER;
OP_SYS: PROCEDURE OPTIONS(MAIN):

DECLARE ( PAGE_SZE, /* THE PAGE SIZE. */
MX_PGS, /* MAX. NO. OF INSTR. PAGES POSSIBLE. */
N,R, /* # OF SCR'S, # OF INSTR NODES. */
CUR_SZE,PAGE,PAGE1,PAGE2,NODE1,SCR,SCR_,NXT,NODE,R1,
I,J,J1,J2,K /* TEMPORARIES. */
YES_HIT(I), /* A TEMPORARY INDICATOR. */
ND_SZE(N), /* SIZE OF CUR. NODE. */
PG_SZE(MX_PGS), /* SIZE OF CUR. PAGE. */
REORDERED_SET(N), /* SET OF REORDERED NODES. */
REORD_INDX(N), /* INDEX INTO REORDERED_SET. */
ENTRY_ND(I), /* ENTRY NODES FOR THE SCR. */
EXIT_ND(I), /* EXIT NODES FOR THE SCR. */
(S CR_N D_ MA P(R) BIT(R)), /* SCR TO NODE MAP. */
PG_ND_MAP(MX_PGS) BIT(N), /* PAGE TO NODE MAP. */
ND_PG_MAP(N) HIT(MX_PGS) /* NODE TO PAGE MAP. */
) CONTROLLED,
ZERO BIT(100) INITIAL (100)*0*B);}

GET LIST (N,R,PAGE_SZE,MAX_PGS):
R1 = R + N;
ALLOCATE ND_SZE,PG_SZE,REORDERED_SET,REORD_INDX,ENTRY_ND,EXIT_ND;
ALLOCATE SCR_ND_MAP,PG_ND_MAP,ND_PG_MAP;
GET SKIP LIST(REORDERED_SET);
GET SKIP LIST(ENTRY_ND);
GET SKIP LIST(EXIT_ND);
GET SKIP LIST(ND_SZE);
GET SKIP EDIT(SCR_ND_MAP) ((R) (COL(1),B(R1)));
PG_ND_MAP = SUBSTR(ZERO,I,N);
ND_PG_MAP = SUBSTR(ZERO,I,MAX_PGS);
DO I = 1 TO N;
J = REORDERED_SET(I) ; REORD_INDX(J) = I; END;
PG_SZE = 0;
CALL PAGINATE;
CALL PRINT_LIST;

PAGINATE: PROCEDURE:

CUR_SZE = 0;
PAG E = 0;
DO SCR_# = 1 TO R;
/* SET J1 & J2 TO BOUNDARY NODES OF SCR_. */
J1 = REORD_INDX(ENTRY_ND(SCR_#));
J2 = REORD_INDX(EXIT_ND(SCR_#));
/* CHECK IF SCR_# CONTAINS ANY NESTED SCR'S. IF SO THEN */
/ compact the pages of the scr_# by filling in holes in the /*
* entry & exit pages of the nested scr's. ---------- */
call chk_scr ; nxt = -1 ;
if yes then call compact_scr ;
/* paginate the unassigned nodes of scr_#. ---------- */
call assign_nos ;
end ;

chk_scr: procedure;
/* a procedure to check if scr_# contains a nested scr. ------- */
yes = '1'b ;
scp = scr_# - 1 ;
do while (scp > 0) ;
n1 = scr + n ;
if substr(scr_nd_map(scr_#),n1,1) = '1'b then return ;
scp = scp - 1 ;
end ;
yes = '0'b ;
return ;
end chk_scr ;

compact_scr: procedure ;
/* check if two or more "adjacent" scr's can be compacted . -- */
i = j1 ;
do while (i > j2) ;
node = reordered_set(i) ;
if nd_size(node) = 0 then do ;
call find_page(node,page1) ;
if page1 > 0 then yes = '1'b ; else yes = '0'b ;
do while (yes) ;
i1 = i + 1 ;
if i1 > j2 then i1 = j1 ;
yes = '0'b ;
node1 = reordered_set(i1) ;
if nd_size(node1) = 0 then do ;
call find_page(node1,page2) ;
if page1 = page2 then do ;
if pg_size(page1) + pg_size(page2) > page_size then do ;
call merge_pgs ; yes = '1'b ;
end ;
end ;
end ;
end ;
i = i + 1 ;
end ;
/* FILL ENTRY & EXIT PAGES OF NESTED SCR'S. */

DO J = (N+1) TO (M+SCR_#-1);
   IF SUBSTR(SCR_ND_MAP(SCR_#,J),1)='1B THEN DO;
      /* SCR J IS NESTED IN SCR_#. COMPACT ITS ENTRY/EXIT PG'S. */
      NODE = ENTRY_ND(J-N); NXT = -1; CALL PG_COMPACT;
      NODE = EXIT_ND(J-N); NXT = 1; CALL PG_COMPACT;
   END;
END;
RETURN;
END COMPACT_SCP;

PG_COMPACT: PROCEDURE:
/* COMPACT THE ENTRY(EXIT) PAGES OF A NESTED SCR WITH NODES */
/* PRECEDING(FOLLOWING) THE ENTRY(EXIT) NODE OF NESTED SCR. */
CALL FIND_PAGE(NODE,PAGE);
I = KEVD,IMXI(NODE);
DO WHILE (PG_SLE(PAGE)<PAGE_SIZE);
   I = I + NXT;
   IF I<J1 THEN I=J2;
   IF I>J2 THEN I=J1;
   NODE = REORDERED_SET(I);
   /* CHECK TO SEE IF NODE IS AN SCR. */
   IF NO_SIZE(NODE)=0 THEN RETURN;
   /* NODE IS NOT CONTAINED IN AN SCR. MERGE (PART OR WHOLE OF */
   /* THIS) NODE INTO ENTRY/EXIT PAGE OF THE SCR. */
   SUBSTR(PG_NN_MAP(PAGE),NODE,1) = '1B;
   SUBSTR(PG_NN_MAP(PAGE),NODE,1) = '1B;
   PG_SIZE(PAGE) = PG_SIZE(PAGE) + NO_SIZE(NODE);
   IF PG_SIZE(PAGE)>PAGE_SIZE THEN DO;
      NO_SIZE(NODE) = PG_SIZE(PAGE) - PAGE_SIZE;
      PG_SIZE(PAGE) = PAGE_SIZE;
   END; ELSE NO_SIZE(NODE) = 0;
END;
RETURN;
END PG_COMPACT;

ASSIGN_NDS: PROCEDURE:
/* ASSIGN NODES OF SCR_# TO PAGES. IF SCR_# CONTAINS A NESTED */
/* SCR THEN PAGINATE NODES BETWEEN ENTRY NODE OF SCR_# & ENTRY */
/* NODE OF NESTED SCR BACKWARDS SO THAT EMPTY SPACE, IF ANY, IS */
/* LEFT ON THE ENTRY PAGE FOR SCR_#. */
I = J1;
CALL CHK_SCR;
NODE = REORDERED_SET(I);
IF NO_SIZE(NODE)=0 THEN DO;
IF YES THEN DO:
    /* PAGINATE NODES BACKWARDS FROM FIRST SCR TO ENTRY NODE */
    DO WHILE (NO_SIZE(NODE)=0):
        I = I + 1;
        NODE = REORDERED_SET(I);
        END;
    J = I - 1;
    DO J TO J1 BY -1;
        NODE = REORDERED_SET(I);
        CALL ASSIGN_PG;
        END;
    I = J + 1;
    END;
END;
DO WHILE (I>J2):
    /* DO THE FOLLOWING FOR ALL NODES BELONGING TO THE SCR. */
    NODE = REORDERED_SET(I);
    IF NO_SIZE(NODE)=0 THEN CALL ASSIGN_PG; ELSE CALL NEW_PAGE;
    END;
CALL NEW_PAGE;
/* PUT NODES ON ENTRY & EXIT PAGES TOGETHER IF COMBINED SIZE */
/* DOES NOT EXCEED PAGE_SIZE. */
NODE = ENTRY_NO(SRC_#); CALL FIND_PAGE(NODE,PAGE1);
NODE1 = EXIT_NO(SRC_#); CALL FIND_PAGE(NODE1,PAGE2);
IF PG_SIZE(PAGE1)+PG_SIZE(PAGE2)>PAGE_SIZE THEN CALL MERGE_PGS;
ASSIGN_PG: PROCEDURE;
    /* ASSIGN 'NODE' TO A PAGE. */
    IF CUR_SIZE=0 THEN PAGE=PAGE+1;
    CUR_SIZE=CUR_SIZE+NO_SIZE(NODE);
    ND_SIZE(NODE) = 0;
    SUBSTR(PG_ND_MAP(PAGE),NODE,1) = '1'B;
    SUBSTR(ND_PG_MAP(NODE),PAGE,1) = '1'B;
    DO WHILE (CUR_SIZE>Page_SIZE);
        PG_SIZE(PAGE) = PAGE_SIZE;
        CUR_SIZE = CUR_SIZE - PAGE_SIZE;
        IF CUR_SIZE>0 THEN DO:
            PAGE = PAGE + 1;
            SUBSTR(PG_ND_MAP(PAGE),NODE,1) = '1'B;
            SUBSTR(ND_PG_MAP(NODE),PAGE,1) = '1'B;
        END;
        END;
    END;
RETURN;
END ASSIGN_PG;
NEW_PAGE: PROCEDURE;
/* START A NEW PAGE. ---------------------------------------- */
  IF CUR_SIZE>0 THEN DO;
    PG_SZF(PAGE) = CUR_SIZE;
    CUR_SIZE = 0;
  END;
  RETURN;
END NEW_PAGE;
RETURN;
END ASSIGN_NDS;

MERGE_PGS: PROCEDURE;
/* MERGE NODE1 AND NODE ON TO SAME PAGE. --------------------- */
  PG_SIZE(PAGE1) = PG_SIZE(PAGE1) + PG_SIZE(PAGE2);
  PG_SIZE(PAGE2) = 0;
  SUBSTR(PG_MAP(NODE),PAGE2,1)="1"B;
  SUBSTR(PG_MAP(NODE1),PAGE2,1)="0"B;
  SUBSTR(PG_MAP(NODE1),NODE1,1)="1"B;
  SUBSTR(PG_MAP(PAGE2),NODE1,1)="0"B;
  RETURN;
END MERGE_PGS;

FIND_PAGE: PROCEDURE(NODE,II);
DECLARE (NODE,I,II) FIXED BINARY(31,0);
  II= 0;
DO I = 1 TO MX_PGS;
  IF SUBSTR(PG_MAP(I),NODE,1)="1"B THEN DO;
    II = I;
    IF PG_SIZE(II)<PAGE_SIZE THEN RETURN;
  END;
END;
RETURN;
END FND_PAGE;
RETURN;
END PAGINATE;

PRINT_LIST: PROCEDURE;
  PUT SKIP EDIT(' PAGE_# PAGE_SIZE PAGE_MEMBERSHIP.')(A);
  PUT SKIP;
  DO K = 1 TO MX_PGS;
    IF PG_MAP(K)=SUBSTR(ZERO,1,N) THEN
      PUT SKIP EDIT(K,PAGE_SIZE(K),PG_MAP(K))
       (F(10),F(10),COL(25),B(10));
  END;
  RETURN;
END PRINT_LIST;
END OP_SYS;
CASE_I: PROCEDURE OPTIONS(MAIN);

DECLARE { LEVEL, /* INDICATES THE NESTING_LVL OF ROUTINE */
PAGE_SZF,N,R, /* N=# OF NODES, R=# OF W.S.'S */
INITIAL,FIALN,IN_TR_CHNG,J1,STOP,II,PGS,
I,J,K,L,M,WII /* TEMPORARIES */
} FIXED BINARY(31,0);

DECLARE { REORDERED_SET(N), /* INITIALLY CONTAINS THE UNORDERED SET */
NDS_IN_PG(PG5), /* NO. OF NODES IN A PAGE */
TOU(R,R),
TOU(R,P),TOU2(R,R) /* THE W.S. INTER-REFERENCE FREQ MTRX */
} FIXED BINARY(31,0) CONTROLLED;

DECLARE { TEMP BIT(R) VARYING CONTROLLED, /* TEMP */
TEMP1 BIT(N) CONTROLLED, /* BIT */
Q1(R1) BIT(N) VARYING CONTROLLED, /* ARRAYS */
WS(N) BIT(R) CONTROLLED; /* WS MEMBERSHIP OF NDS */

GET LIST (N,R,PAGE_SZF);
R1 = (R+1) / 2;
PGS = (N+PAGE_SZF-1)/PAGE_SZF;
ALLOCATE REORDERED_SET,TOU,TOU1,TOU2,TEMP,TEMP1,QI,WS,NDS_IN_PG;
GET SKIP LIST (REORDERED_SET);
GET SKIP LIST (TOU);
PUT SKIP EDIT* TOU -- INTER-WS REF. FREQ MATRIX **** *(A);
PUT SKIP EDIT (TOU) ((R) (COL(2),(R) F(4)));
DO I = 1 TO R;
GET SKIP EDIT(TEMP1) (B(N));
DO J = 1 TO N;
SUBSTR(WS(J),I,1) = SUBSTR(TEMP1,J,1);
END;
END;

PUT SKIP EDIT* PAGE SIZE **** = *(PAGE_SZF)(A,F(4));
PUT SKIP EDIT* UNORDERED DATA NODES ******** *(A);
CALL PRINT_LIST;
PUT SKIP(2);
TOU1 = TOU;
TOU2 = TOU;

INITIAL = 1;
FINAL = 1;
DO WHILE (FINAL<CN);
LEVEL = 0;
QI = **3;
CALL FIND_IJ(J1,J,TOU);
IF STOP = 1 THEN GO TO PRNT;
K = INITIAL;
L = N;
CALL COMPACT(J,K,L,FINAL);
TOU(*,J) = 0 : IF FINAL-INITIAL>PAGE_SZE THEN DO :
    I1 = I ; J1 = J ;
    TOU1 = TOU ;
    TOU1(J,*#) = 0 ; TOU1(*,I) = 0 ; TOU1(I,*#) = 0 ;
    CALL REORDER(INITIAL,FINAL,I1,J1,TOU1) ;
END :
    INITIAL = FINAL + 1 :
END :
PRINT: /* PRINT THE ORDERED SET OF DATA NODES ------------------------ */
PUT SKIP EDIT(' ORDERED DATA NODES ........ ')(A) ;
CALL PRINT_LIST ;

REORDER: PROCEDURE (INITIAL,FINAL,I,J,TOU1) RECURSIVE ;
DECLARE TOU1(R,R) FIXED BINARY(31,0) CONTROLLED ;
DECLARE ( INITIAL,FINAL,FINAL1,TOU1(*,*,I,J,K,L,I1,J1 )
FIXED BINARY(31,0) ; /* RECURSIVE PROC WHICH REORDERS NODES BETWEEN INITIAL & FINAL
ACCORDING TO CASE 1 DATA ORDERING METHOD, BY DISTRIBUTING
NODES REF. BY WS I EQUALLY AMONG DATA PAGES OF WS J, AND BY
FINDING THE NEXT TWO WS'S FOR WHICH NODES SHOULD BE ORDERED
WITHIN THIS SET OF DATA NODES. ----------------------------- */
LEVEL = LEVEL + 1 ;
CALL DISTRIBUTE(I,INITIAL,FINAL) ;
FINAL1 = INITIAL ;
DO WHILE (FINAL1<FINAL) ;
    CALL FIND_IJ1(I,J,TOU1) ; IF STOP=1 THEN GO TO QUIT :
    K = INITIAL ;
    L = FINAL ;
    CALL COMPACT(J,K,L,FINAL1) ;
    TOU1(*,J) = 0 ; IF FINAL1-INITIAL>PAGE_SZE THEN DO :
        ALLOCATE TOU :
        TOU = TOU1 ;
        I1 = I ; J1 = J ;
        TOU1(J,*#) = 0 ; TOU1(*,I) = 0 ; TOU1(I,*#) = 0 ;
        CALL REORDER(INITIAL,FINAL1,I1,J1,TOU1) ;
        FREE TOU :
    END :
    INITIAL = FINAL1 + 1 :
END :
QUIT: QI(LEVEL) = '"R' ;
LEVEL = LEVEL - 1 ;
RETURN :
END REORDER ;
COMPACT: PROCEDURE (J, Jl, K1, FINAL) ;
DECLARE (J, I, K, FINAL, Jl, K1, il, 12) FIXED BINARY(31, 0) ;
/* PROCEDURE FOR PUTTING TOGETHER ALL NODES BETWEEN PNTRS J1 & */
/* K1, REFERENCED BY WS J. SETS FINAL = LAST NODE REF BY WS J */
FINAL = K1 ;
DO I1 = J1 TO (K1-1) ;
I = REORDERED_SET(I1) ;
IF SUBSTR(WS(I), J, 1) = '0'B THEN DO ;
DO I2 = K1 TO (I1+1) BY -1 ;
K = REORDERED_SET(I2) ;
IF SUBSTR(WS(K), J, 1) = '1'B THEN DO ;
CALL CHANGE_ELEMENTS(I1, I2) ;
IF INTR_CHNG = 1 THEN FINAL = I1 ;
END ;
END ; ELSE FINAL = I1 ;
END ;
RETURN ;
END COMPACT ;

CHANGE_ELEMENTS: PROCEDURE (J, K) ;
DECLARE (J, Jl, K, K1, L) FIXED BINARY(31, 0) ;
/* INTEP CHANGE DATA NODES J & K IF PRIOR REORDERINGS ARE NOT */
/* UPSET BY SUCH A CHANGE. THIS IS DONE BY CHECKING QI VECTORS */
/* WHICH RECORD PRIOR ORDERINGS. SETS INTR_CHNG=1 IF THE CHANGE*/
/* IS EFFECTED. ----------------------------------------------- */
L = LEVEL ;
INTR_CHNG = 0 ;
J1 = REORDERED_SET(J) ;
K1 = REORDERED_SET(K) ;
DO WHILE (L > 0) ;
IF SUBSTR(QI(L), J1, 1) = SUBSTR(QI(L), K1, 1) THEN RETURN ;
L = L - 1 ;
END ;
INTR_CHNG = 1 ;
M = REORDERED_SET(J) ;
REORDERED_SET(J) = REORDERED_SET(K) ;
REORDERED_SET(K) = M ;
RETURN ;
END CHANGE_ELEMENTS ;

FIND_IJ: PROCEDURE (I, J, TOU) ;
/* FIND THE LARGEST ELEMENT IN TOU MATRIX & SET I, J ----------- */
DECLARE (I, J, K, L, MAX, TOU(*,*)) FIXED BINARY(31, 0) ;
MAX = 0 ;
DO K = 1 TO R ;
DO L = 1 TO R ;
IF T0U(K,L) > MAX THEN DO ;
    I = K ;
    J = L ;
    MAX = T0U(I,L) ;
END ;
END ;
IF MAX = 0 THEN STOP = 1 ; ELSE STOP = 0 ;
RETURN ;
END FIND_IJ ;

DISTRIBUTE: PROCEDURE(I,INITIAL,FINAL) ;

/* ORDER NODES SO THAT EACH PAGE CONTAINS AN EQUAL NO. OF NODES */
/* REFERENCED BY W.S. 1.  ALSO CHECK THAT THIS REORDERING DOES */
/* NOT UPSERT PRIOR REORDERINGS. */
DECLARE ( I, INITIAL, FINAL, J, K, L, M, I1, I2, I3, I4, I5, KOUNT, MIN_NDS,
    MAX_NDS, J1, J2, J3, K1, K2, K3, J4, K4 ) FIXED BINARY(31,0) ;

KOUNT = 0 ;
M = 0 ;
NDS_IN_PG = 0 ;
/* SET QI VECTOR & COUNT NO OF NODES IN EACH PAGE. */
DO II = 1 TO N ;
    QI(LEVEL) = QI(LEVEL) II '0'B ;
END ;

DO II = INITIAL TO FINAL ;
    II2 = REORDERED_SET(II) ;
    SUBSTRING(LEVEL),I2,1) = SUBSTRING(IS(II),II,II) ;
    IF SUBSTRING(IS(II2),II,II)='1'B THEN KOUNT=KOUNT+1 ;
    IF MOD(FINAL,PAGE_SIZE)=0 THEN DO ;
        M = M + 1 ;
        NDS_IN_PG(M) = KOUNT ;
        KOUNT = 0 ;
    END ;
END ;

KOUNT = 0 ;
DO II = 1 TO M ;
    KOUNT = KOUNT + NDS_IN_PG(II) ;
END ;

/* COUNT THE TOTAL NO OF ELEMENTS TO BE DISTRIBUTED. */
KOUNT = 0 ;
DO II = INITIAL TO FINAL ;
    II2 = REORDERED_SET(II) ;
    SUBSTRING(LEVEL),I2,1) = SUBSTRING(IS(II),II,II) ;
    IF SUBSTRING(IS(II2),II,II)='1'B THEN KOUNT=KOUNT+1 ;
    IF MOD(FINAL,PAGE_SIZE)=0 THEN DO ;
        M = M + 1 ;
        NDS_IN_PG(M) = KOUNT ;
        KOUNT = 0 ;
    END ;
END ;

/* NO ORDERING NEEDED IF ONLY ONE DATA PAGE. */
IF (M=11)(KOUNT<2) THEN RETURN ;
MIN_NDS = KOUNT / PAGE_SIZE ;
MAX_NDS = (KOUNT+PAGE_SIZE-1) / PAGE_SIZE ;
DO I1 = 1 TO M ;
   NXT: /* LOOK FOR A PAGE WITH NODS LESS THAN MIN NODES. */
   IF NDS_IN_PG(I1)<MIN_NDS THEN DO ;
      DO I2 = 1 TO M ;
         IF NDS_IN_PG(I2)<MIN_NDS THEN DO ;
            /* FIND TWO NODFS TO SWAP IN PGS I1 & I2. */
            CALL SWAP(I1,I2) ;
            IF INTR_CHNG=1 THEN DO ;
               NDS_IN_PG(I1) = NDS_IN_PG(I1) + 1 ;
               NDS_IN_PG(I2) = NDS_IN_PG(I2) - 1 ;
               GO TO NXT ;
            END ;
         END ;
      END ;
   END ;
   END ;
   NXTT: /* LOOK FOR PGS WITH NODES > MAX NODES. */
   IF NDS_IN_PG(I1)>MAX_NDS THEN DO ;
      DO I2 = 1 TO M ;
         IF NDS_IN_PG(I2)<MAX_NDS THEN DO ;
            /* FIND TWO NODFS TO SWAP IN PAGES I2 & I1. */
            CALL SWAP(I2,I1) ;
            IF INTR_CHNG=1 THEN DO ;
               NDS_IN_PG(I1) = NDS_IN_PG(I1) + 1 ;
               NDS_IN_PG(I2) = NDS_IN_PG(I2) - 1 ;
               GO TO NXTT ;
            END ;
         END ;
      END ;
   END ;
   END ;

SWAP: PROCEDURE(I1,I2) :
   DECLARE(I1,I2) FIXED BINARY(31,0) ;
   /* ADD A NODE WITH '1' IN WS I TO PAGE I1 & REMOVE THIS NODE */
   /* FROM I2. INTR_CHNG=0 IF THIS IS NOT POSSIBLE. */
   J = INITIAL - MOD(INITIAL,PAGE_SIZE) ;
   J1 = (I1-1)*PAGE_SIZE + J ; IF J1<INITIAL THEN J1=INITIAL ;
   J2 = J1 + PAGE_SIZE - 1 ; IF J2>FINAL THEN J2=FINAL ;
   K1 = (I2-1)*PAGE_SIZE + J ; IF K1<INITIAL THEN K1=INITIAL ;
   K2 = K1 + PAGE_SIZE - 1 ; IF. K2>FINAL THEN K2=FINAL ;
   DO J3 = J1 TO J2 ;
      J4 = REORDERED_SET(J3) ;
      IF SUBSTR(WS(J4),I,1)='0' THEN DO ;
         DO K3 = K1 TO K2 ;
            K4 = REORDERED_SET(K3) ;
            IF SUBSTR(WS(K4),I,1)='1' THEN DO ;
               CALL CHANGE_ELEMENTS(J3,K3) ;
IF INTR_CHNG = 1 RETURN
END;

END;
END;
END;

INTR_CHNG = 0 RETURN;
ENn SWAP RETURN;
END DISTRIBUTE
PRINT_LIST: PROCEDURE
DECLARE (I,J,Jl) FIXED BINARY(31,0)
DO I = 1 TO R;
on J = 1 TO N
Jl = REORDERED_SF.T(J) ;
SUBSTR(TEMP1,J,11 = SUBSTR(WS(J1I,I,1)
END
PUT SKIP EDIT (TEMP1) (COL(101,BIN)) ;
END PRINT_LIST ;
REORDER: PROCEDURE OPTIONS(MAIN);

/*  
PROCEDURE FOR REORDERING OF DATA NODES
*/

DECLARE ( X, /* THE DIFFERENCE IN SET S1 & S2 */
  MIN_X, /* THE MIN VALUE OF X SO FAR */
  TTL_NODES, /* TOTAL NODES TO BE REORDERED */
  MX_WS_, /* TOTAL WORKING SETS */
  MX_RANGE, /* MAX POSSIBLE NODES FOR MX_WS_ */
  DATA_NODE_S, /* DATA NODES REMAINING TO BE REORDERED */
  DATA_NODE_PTR, /* DATA NODE # BEING ORDERED */
  CURR_NODE, /* NO OF NODES ORDERED */
  S1, S2, /* THE SETS OF W.S.'S */
  I, J, K, L /* TEMPORARIES */
) FIXED BINARY (31,0);

DECLARE ( DATA_NODES(TTL_NODES), /* LIST OF NODES TO BE ORDERED */
  REORD_LIST(TTL_NODES), /* REORDERED LIST OF NODES */
  OUT(TTL_NODES), /* TEMP FOR PRINTOUT */
) FIXED BINARY (31,0) CONTROLLED;

GET EDIT(X_WS_#, TTL_NODES, DATA_NODE_PTR) (3 F(4)) ;
ALLOCATE DATA_NODES, REORD_LIST, OUT ;
GET SKIP LIST (DATA_NODES) ;
DATA_NODE_S = TTL_NODES ;
MX_NODES = 2**MX_WS_ ;

PUT PAGE ;
PUT SKIP EDIT(' UNORDERED LIST ') (A) ;
REORD_LIST = DATA_NODES ;
CALL PRT_LIST ;
REORD_LIST = 0 ;

S1 = DATA_NODES(DATA_NODE_PTR) ;
REORD_LIST(1) = S1 ;
DATA_NODES(DATA_NODE_PTR) = DATA_NODES(TTL_NODES) ;
DATA_NODE_S = DATA_NODE_S - 1 ;
CURR_NODE = 1 ;

DO WHILE (DATA_NODE_S > 1) :
  CALL NEXT_NODE ; /* FIND NEXT NODE TO BE ORDERED */
  CURR_NODE = CURR_NODE + 1 ;
  REORD_LIST(CURR_NODE) = S1 ;
  DATA_NODES(DATA_NODE_PTR) = DATA_NODES(DATA_NODE_S) ;
  DATA_NODE_S = DATA_NODE_S - 1 ;
END;
REORD_LIST(TTL_NODES) = DATA_NODES(1) ;
PUT SKIP(2) ;
PUT SKIP EDIT(* ORDERED LIST')(A) ;
CALL PRINT_LIST ;

NEXT_NODE: PROCEDURE ;
J = 1 ;
MIN_X = MX_NODES ;
DO I = 1 TO DATA_ND_SIZE ;
S2 = DATA_NODES(I) ;
CALL FINDX(S1,S2,X,MIN_X) ; /* FIND X & COMPARE WITH MIN_X */
IF X=MIN_X THEN J=I ;
END ;
/* J POINTS TO THE NEXT NODE TO BE ORDERED */
S1 = DATA NODES(J) ;
DATA_ND_PTR = J ;
RETURN ;
END NEXT_NODE ;

PRINT_LIST: PROCEDURE ;
DO I = 1 TO MX_WS_# ;
DO J = 1 TO TTL_NODES ;
K = REORD_LIST(J) ;
OUT(J) = MOD(K,2) ;
REORD_LIST(J) = REORD LIST(J) / 2 ;
END ;
PUT SKIP EDIT (OUT) (COLUMN(10),(TTL NODES) F(I)) ;
END ;
RETURN ;
END PRINT_LIST ;

END REORDER ;
* ASSEMBLER SUBROUTINE FINDX
* FINDS THE DATA NODE REFERENCED BY ELEMENTS OF THE SET X OF
* DATA ORDERING ALGORITHM R OF SECTION 2.3
ENTRY FINDX

FINDX  STM 14,12,12(13)  SAVE CALLER’S REGISTERS
       LR 12,15  ESTABLISH PERMANENT ADDRESSIBILITY
       USING FINDX,12  SET UP BASE REG
       ST 13,SA+4  SET UP SA FOR FINDX
       LR 2,13  **
       LA 13,SA  **
       ST 13,8(2)  **
       LM 3,6,0(1)  LOAD S1,S2,X,MIN_X ADDR. RESPLY.

* MAIN PROGRAM STARTS HERE
* L 7,0(3)  S1
L 8,0(4)  S2
XR 7,8  REG MUX HAS X
ST 7,0(5)  STORE X
L 8,0(6)  MIN_X
SLL 8,16  SHIFT LEFT (32-MX_WS_#) TO SAVE TIME
SLL 7,16  **
       L 11=F*16'  MAX NO OF SLL OPERATIONS
LOOP  SLL 7,1  SHIFT LEFT TO COMPARE COMPONENTS **
       SLL 8,1  ** OF X AND MIN_X
       LR 9,7  **
       LR 10,8  **
       N 9=X’80000000’  MASK LAST 31 BITS
       N 10=X’80000000’  **
       O 9=X’00000001’  TO DIFFERENTIATE BETW -0 & +0
       O 10=X’00000001’  **
       CR 9,10  **
       N/E OUT  A DIFFERENCE IN X & MIN_X IS FOUND
       BCT 11,LOOP  O.W. LOOP TILL ALL BITS ARE TESTED
OUT  BL EXIT  X HAS A W.S. COMPONENT>MIN_X
L 7,0(5)  OTHERWISE **
       ST 7,0(6)  ** SET MIN_X=X & RETURN

* RESTORE CALLER’S STATE & RETURN
* EXIT  L 13,SA+4  RESTORE REG13 FOR CALLER’S SA
LM 14,12,12(13)  RESTORE CALLER’S REGISTERS
BR 14  RETURN
SA DS 1BF  END
ON ENOFILE lSYSINI GO TO QUIT;
GET LIST IN, PAGE) ;
ALLOCATE ADJCNCY, RES_SET, TEMP1, TEMP2
GET SKIP EDIT (ADJCNCY, RES_SET IN (COllll, B(NII, COLI11, 8lN))
DO WHILE (1)
    PUT SKIP (2) ;
    PUT SKIP EDIT (' REFERENCED PAGE = PAGE) (A,F(6)) ;
    PUT SKIP EDIT (' RESIDENT SET = RES_SET) (A,COL(25), B(N)) ;
    CALL I_N_R ;
    PUT SKIP EDIT (' PAGE WITH MAX INR = INR_PAGE) (A,F(6)) ;
    PUT SKIP EDIT (' VALUE OF INR = MX_INR) (A,F(6)) ;
    GET SKIP LIST(PAGE) ;
    GET SKIP EDIT (RES_SET) (B(N)) ;
END ; /* OF DO WHILE (1) ---------------------------------------- */
I_N_R: PROCEDURE ;
/* THIS PROCEDURE SELECTS THE PAGE WITH MAX INR TO BE REPLACED */
/* IT ACCEPTS AS INPUT THE CURRENT REF PAGE, THE RESIDENT SET */
/* & THE ADJACENCY MATRIX. ---------------------------------------- */
PATH_LEN = 1 ;
MX_INR = 0 ;
INR_PAGE = 0 ;
TEMP1 = ADJCNCY(PAGE) ;
DO WHILE (PATH_LENCN)
    TEMP2 = SUBSTR(TEMP3,1,N) ;
    DO I = 1 TO N ;
        IF SUBSTR(TEMP1,I,1)='1' THEN DO ;
            /* FIND ALL PAGES AT A DISTANCE OF 1 FROM I' TH PAGE. */
            TEMP2 = TEMP2 | ADJCNCY(I) ;
        END ;
    END ;
END ;
PATH_LEN = PATH_LEN + 1;
/* CHECK IF ANY PAGE ON RES. SET IS IN THE LIST OF PAGES AT */
/* PATH_LEN FROM CURRENT PAGE. --------------------------------- */
TEMP1 = TEMP2;
TEMP2 = TEMP2 & RES_SET;
DO I = 1 TO N;
  IF SUBSTR(TEMP2, I, 1) = '1' THEN DO;
    /* PAGE I HAS A PATH OF LENGTH PATH_LEN FROM CUR. PAGE */
    /* CHECK IF THIS PAGE HAS THE MAX VALUE OF INR, IF SO */
    /* IT IS CURRENT CANDIDATE FOR REPLACEMENT. ---------- */
    IF PATH_LEN > MX_INR THEN DO;
      MX_INR = PATH_LEN;
      INR_PAGE = I;
    END;
    /* REMOVE I' TH PAGE FROM RES. SET SINCE IT'S INR HAS */
    /* BEEN ASCERTAINED. ------------------------------- */
    SUBSTR(RES_SET, I, 1) = '0';
  END;
END;
END;
DO I = 1 TO N;
  IF SUBSTR(PES_SET, I, 1) = '1' THEN DO;
    /* PAGE I IS NOT RELEASABLE FROM 'PAGE' HENCE REPLACE THIS PAGE */
    MX_INR = 10000;
    INR_PAGE = I;
    RETURN;
  END;
END;
RETURN;
END I_N_R;
QUIT:
END OP_SYS;
REFERENCES


